

Max-Sum

Generated by Doxygen 1.8.1.2

Thu Aug 2 2012 17:47:11

Contents

1	Main Page	1
1.1	Overview	1
1.2	Library Organisation	1
1.3	The DiscreteFunction Class	1
1.3.1	Construction	2
1.3.2	Element Access	2
1.3.3	Mathematical Operations	4
1.3.3.1	Scalar Operations	4
1.3.3.2	Domain Operations	4
1.3.3.3	Arithmetic Operations	5
1.3.4	Comparison Operations	6
1.3.5	Miscellaneous Operations	7
1.4	The DomainIterator Class	7
1.5	The MaxSumController Class	7
1.6	Tips on Writing Efficient Code	8
1.7	Future Work	9
2	Namespace Index	11
2.1	Namespace List	11
3	Class Index	13
3.1	Class List	13
4	File Index	15
4.1	File List	15
5	Namespace Documentation	17
5.1	maxsum Namespace Reference	17
5.1.1	Detailed Description	19
5.1.2	Typedef Documentation	19
5.1.2.1	FactorID	19
5.1.2.2	ValIndex	19
5.1.2.3	ValType	19

5.1.2.4	VarID	20
5.1.3	Function Documentation	20
5.1.3.1	allRegistered	20
5.1.3.2	equalWithinTolerance	20
5.1.3.3	getDomainSize	20
5.1.3.4	getNumOfRegisteredVariables	21
5.1.3.5	ind2sub	21
5.1.3.6	isRegistered	21
5.1.3.7	marginal	22
5.1.3.8	maxMarginal	23
5.1.3.9	meanMarginal	23
5.1.3.10	minMarginal	24
5.1.3.11	operator!=	25
5.1.3.12	operator<<	25
5.1.3.13	operator==	25
5.1.3.14	registerVariable	25
5.1.3.15	registerVariables	26
5.1.3.16	sameDomain	26
5.1.3.17	strictlyEqualWithinTolerance	27
5.1.3.18	sub2ind	27
5.1.3.19	sub2ind	27
5.1.4	Variable Documentation	28
5.1.4.1	DEFAULT_VALUE_TOLERANCE	28
5.2	maxsum::util Namespace Reference	28
5.2.1	Detailed Description	28
6	Class Documentation	29
6.1	maxsum::BadDomainException Class Reference	29
6.1.1	Detailed Description	29
6.1.2	Constructor & Destructor Documentation	29
6.1.2.1	BadDomainException	29
6.1.3	Member Function Documentation	30
6.1.3.1	what	30
6.2	maxsum::util::KeySet< Map >::const_iterator Class Reference	30
6.2.1	Detailed Description	30
6.3	maxsum::DiscreteFunction Class Reference	30
6.3.1	Detailed Description	33
6.3.2	Constructor & Destructor Documentation	33
6.3.2.1	DiscreteFunction	33
6.3.2.2	DiscreteFunction	33

6.3.2.3	DiscreteFunction	34
6.3.2.4	DiscreteFunction	34
6.3.3	Member Function Documentation	34
6.3.3.1	add	34
6.3.3.2	at	34
6.3.3.3	at	35
6.3.3.4	at	35
6.3.3.5	at	35
6.3.3.6	condition	36
6.3.3.7	dependsOn	36
6.3.3.8	expand	36
6.3.3.9	expand	36
6.3.3.10	expand	37
6.3.3.11	maxnorm	37
6.3.3.12	min	37
6.3.3.13	operator()	37
6.3.3.14	operator()	37
6.3.3.15	operator()	38
6.3.3.16	operator()	38
6.3.3.17	operator()	38
6.3.3.18	operator()	38
6.3.3.19	operator()	39
6.3.3.20	operator=	39
6.3.3.21	operator=	39
6.3.3.22	sizeBegin	40
6.3.3.23	sizeEnd	40
6.3.3.24	swap	40
6.3.3.25	varBegin	40
6.3.3.26	varEnd	40
6.3.4	Friends And Related Function Documentation	40
6.3.4.1	operator<<	40
6.4	maxsum::DomainConflictException Class Reference	41
6.4.1	Detailed Description	41
6.4.2	Constructor & Destructor Documentation	41
6.4.2.1	DomainConflictException	41
6.4.3	Member Function Documentation	41
6.4.3.1	what	41
6.5	maxsum::DomainIterator Class Reference	42
6.5.1	Detailed Description	43
6.5.2	Constructor & Destructor Documentation	43

6.5.2.1	DomainIterator	43
6.5.2.2	DomainIterator	43
6.5.2.3	DomainIterator	43
6.5.3	Member Function Documentation	43
6.5.3.1	addVars	43
6.5.3.2	condition	44
6.5.3.3	condition	44
6.5.3.4	getInd	44
6.5.3.5	getSubInd	45
6.5.3.6	hasNext	45
6.5.3.7	operator++	45
6.5.3.8	operator++	45
6.5.3.9	validateRange	46
6.6	maxsum::EmptyNoticeException Class Reference	46
6.6.1	Detailed Description	46
6.6.2	Constructor & Destructor Documentation	47
6.6.2.1	EmptyNoticeException	47
6.6.3	Member Function Documentation	47
6.6.3.1	what	47
6.7	maxsum::InconsistentDomainException Class Reference	47
6.7.1	Detailed Description	47
6.7.2	Constructor & Destructor Documentation	48
6.7.2.1	InconsistentDomainException	48
6.7.3	Member Function Documentation	48
6.7.3.1	what	48
6.8	maxsum::util::KeySet< Map > Class Template Reference	48
6.8.1	Detailed Description	49
6.8.2	Constructor & Destructor Documentation	49
6.8.2.1	KeySet	49
6.8.3	Member Function Documentation	49
6.8.3.1	begin	49
6.8.3.2	end	50
6.8.3.3	find	50
6.8.3.4	size	50
6.9	maxsum::MaxSumController Class Reference	50
6.9.1	Detailed Description	51
6.9.2	Constructor & Destructor Documentation	52
6.9.2.1	MaxSumController	52
6.9.3	Member Function Documentation	52
6.9.3.1	clear	52

6.9.3.2	getFactor	52
6.9.3.3	getValue	52
6.9.3.4	hasEdge	53
6.9.3.5	hasFactor	53
6.9.3.6	hasValue	53
6.9.3.7	optimise	53
6.9.3.8	removeFactor	54
6.9.3.9	setFactor	54
6.10	maxsum::NoSuchElementException Class Reference	54
6.10.1	Detailed Description	55
6.10.2	Constructor & Destructor Documentation	55
6.10.2.1	NoSuchElementException	55
6.10.3	Member Function Documentation	55
6.10.3.1	what	55
6.11	maxsum::OutOfRangeException Class Reference	55
6.11.1	Detailed Description	56
6.11.2	Constructor & Destructor Documentation	56
6.11.2.1	OutOfRangeException	56
6.11.3	Member Function Documentation	56
6.11.3.1	what	56
6.12	maxsum::util::PostOffice< Sender, Receiver, Message > Class Template Reference	56
6.12.1	Detailed Description	58
6.12.2	Member Typedef Documentation	58
6.12.2.1	ReceiverIt	58
6.12.2.2	SenderIt	58
6.12.3	Constructor & Destructor Documentation	59
6.12.3.1	~PostOffice	59
6.12.4	Member Function Documentation	59
6.12.4.1	addEdge	59
6.12.4.2	addEdge	59
6.12.4.3	curlnMsgs	59
6.12.4.4	curOutMsgs	60
6.12.4.5	hasEdge	60
6.12.4.6	notify	60
6.12.4.7	popNotice	60
6.12.4.8	prevInMsgs	61
6.12.4.9	prevOutMsgs	61
6.12.4.10	receiverBegin	61
6.12.4.11	receiverEnd	61
6.12.4.12	removeEdge	62

6.12.4.13	senderBegin	62
6.12.4.14	senderEnd	62
6.12.4.15	swapOutBoxes	62
6.13	maxsum::util::RefMap< Key, Val > Class Template Reference	63
6.13.1	Detailed Description	63
6.13.2	Member Function Documentation	64
6.13.2.1	begin	64
6.13.2.2	end	64
6.13.2.3	operator[]	64
6.14	maxsum::UnknownAddressException Class Reference	65
6.14.1	Detailed Description	65
6.14.2	Constructor & Destructor Documentation	65
6.14.2.1	UnknownAddressException	65
6.14.3	Member Function Documentation	65
6.14.3.1	what	65
6.15	maxsum::UnknownVariableException Class Reference	66
6.15.1	Detailed Description	66
6.15.2	Constructor & Destructor Documentation	66
6.15.2.1	UnknownVariableException	66
6.15.3	Member Function Documentation	66
6.15.3.1	what	66
7	File Documentation	67
7.1	include/common.h File Reference	67
7.1.1	Detailed Description	68
7.2	include/DiscreteFunction.h File Reference	68
7.2.1	Detailed Description	69
7.3	include/DomainIterator.h File Reference	69
7.3.1	Detailed Description	69
7.4	include/exceptions.h File Reference	69
7.4.1	Detailed Description	70
7.5	include/mainpage.h File Reference	70
7.5.1	Detailed Description	70
7.6	include/MaxSumController.h File Reference	71
7.6.1	Detailed Description	71
7.7	include/PostOffice.h File Reference	71
7.7.1	Detailed Description	72
7.8	include/register.h File Reference	72
7.8.1	Detailed Description	73
7.9	include/util_containers.h File Reference	73

7.9.1 Detailed Description	73
7.10 src/DiscreteFunction.cpp File Reference	73
7.10.1 Detailed Description	74
7.11 src/DomainIterator.cpp File Reference	74
7.11.1 Detailed Description	74
7.12 src/MaxSumController.cpp File Reference	74
7.12.1 Detailed Description	74
7.13 src/register.cpp File Reference	74
7.13.1 Detailed Description	74

Chapter 1

Main Page

1.1 Overview

This library provides an implementation of the max-sum algorithm in C++. For more details about the purpose and theory behind this algorithm, please see <http://eprints.soton.ac.uk/265159/>

The source code for this library can be downloaded from [here](#).

If you have any comments or suggestions about this library or its documentation, please let me know via the main [project website](#).

1.2 Library Organisation

The contents of this library are organised into two namespaces:

- `maxsum`, which contains all functions and classes that form part the public interface to this library; and
- `maxsum::util`, which contains utility code that forms part of the library implementation.

Our intention here is that only functions and types that are of direct interest to third party developers should be located in the `maxsum` namespace. In contrast, those in the `maxsum::util` interface are not intended to form part of the public interface to the library, and so should only be of interest to developers who wish to extend or modify the library.

Of the classes in the `maxsum` namespace, three provide the bulk of the library's functionality, and should be of particular interest to anyone wishing to apply the max-sum algorithm:

- `maxsum::DiscreteFunction`, which is used to represent mathematical functions that depend on the cartesian product of a set of variables with finite domains;
- `maxsum::DomainIterator`, which provides methods for iterating over the domain of `DiscreteFunction` objects; and
- `maxsum::MaxSumController`, which operates on a *factor graph*, to optimise the values assigned to a set of (action) variables.

The following sections describe each of these classes in more detail, together with example extracts of code to demonstrate their proper use.

1.3 The DiscreteFunction Class

The `DiscreteFunction` class is the main workhorse of the `maxsum` library, which not only provides a way to represent mathematical functions, but also to manipulate their values, and combine them using various mathematical

operators to form new functions. As the class name suggests, the main limitation is that only functions with a finite domain can be represented, values of which are stored internally for each possible value of a function's domain.

In future versions, we may provide other classes which can be used to represent functions with continuous domains. However, for now, the [DiscreteFunction](#) class provides all the operations necessary to apply the max-sum algorithm to finite domains. In particular, the following subsections describe the main operations that can be performed using this class.

1.3.1 Construction

The domain of each [DiscreteFunction](#) is specified by a set of variables, identified by ids of type [maxsum::VarID](#). Each variable, k , is associated with a fixed domain size, N_k , which must be registered before the variable is used, and must remain the same through a program's execution. Once registered, a variable can then take on any integer value in the range $[0, N_k]$. For example, to create a [DiscreteFunction](#) that depends on a single variable, 2, we may execute the following code:

```
maxsum::registerVariable (2, 10);
maxsum::DiscreteFunction func(2, 3.2);
```

Here, variable 2 is registered with domain size 10, which means that it can take on values in the range $[0, 9]$. The 2nd parameter of this constructor is used to initialise all values for this function to a specific value, in this case 3.2. So, for example, we have

```
num = func.noVars(); // num == 1 (depends on one variable only)
siz = func.domainSize(); // siz == 10
val = func(0); // val == 3.2
...
val = func(9); // val == 3.2
val = func(10); // result undefined!!
```

In addition to depending on a single value, [DiscreteFunction](#)'s can also depend on multiple variables, or the empty set. In the latter, case the function encapsulates a single scalar constant, which does not depend on any variable, and is index by 0:

```
maxsum::DiscreteFunction func(4.5); // depends on no variables
num = func.noVars(); // num==0 (depends on no variables)
siz = func.domainSize(); // siz==1 (has only one value)
val = func(0); // val == 4.5
val = func(1); // result undefined!!
```

For the former case, there are several options for defining functions that depend on several variables. In particular, we can achieve this by passing a list of variable ids to the constructor using iterators. For example:

```
maxsum::VarID vars[] = {1,4,8}; // array of size 3
maxsum::ValIndex siz[] = {5,10,15}; // var 1 has domain size 5 etc.
maxsum::registerVariables (vars, vars+3, siz, siz+3);
maxsum::DiscreteFunction func(vars, vars+3); // values initialised to 0
num = func.noVars(); // num=3 (depends on three variables)
siz = func.domainSize(); // siz=5*10*15 (size of cartesian product)
val = func(0); // val == 0
val = func(749); // val == 0 (see below for full explanation)
val = func(750); // result undefined!!
```

In addition to these, other constructors include a copy constructor and matching assignment operator; and a default constructor, which creates a function that depends on no variables and has a single value of 0. For details, see the manual page for [DiscreteFunction](#).

1.3.2 Element Access

As demonstrated in the examples above, the basic and most efficient way to access the values of a function, is to pass a single integer value to the overloaded `()` operator. This single integer value acts like a linear index for

N-D arrays in matlab, except that values start from 0 rather than 1. Moreover, all versions of the `()` operator return references to values in a [DiscreteFunction](#), and so can be used for setting as well as reading values. For example, the following code is valid.

```
func(1) = 3.45; // set func(1) to 3.45
val = func(1); // val == 3.45
```

As in matlab, or indeed any scheme for indexing N-D arrays, these linear indices depend on the order in which each dimension is stored in memory. In our case, each variable in a function's domain corresponds to a dimension in an N-D array, with variables ordered from least to most significant according to their ID. For example, if `func` depends on variables 3 and 5 with corresponding domain sizes 2 and 3, then we have the following correspondence between variable values and linear indices:

Variable 3	Variable 5	Linear Index
0	0	0
1	0	1
0	1	2
1	1	3
0	2	4
1	2	5

Here, notice how variable 3, having the smallest variable id, is incremented first, before the variable with the next largest id, which is 5. The situation is the same for variables with more than two variables: the variable with the smallest id is incremented first, and so on until the variable with the highest id, which is incremented last.

Although the most efficient way to access elements using linear indices, these are often not available, and so must be calculated from a set of subindices, relating to named variables. For this reason, several other versions of the `()` operator are available that except subindices. For example, if variable names and values are stored in lists, s.t. the *k*th index gives the value for the *k*th variable, then we can access elements as follows.

```
maxsum::VarID vars[] = {1,4,8}; // array of size 3
maxsum::ValIndex siz[] = {5,10,15}; // var 1 has domain size 5 etc.
maxsum::registerVariables (vars,vars+3,siz,size+3);
maxsum::DiscreteFunction func(vars,vars+3); // values initialised to 0

std::vector<VarID> varVec(vars,vars+3); // copy ids into vector
std::vector<ValIndex> valVec(3); // vector of size 3
valVec[0] = 2; // index value for variable 1
valVec[1] = 4; // index value for variable 4
valVec[2] = 5; // index value for variable 8

// THE FOLLOWING STATEMENTS ALL ACCESS THE SAME ELEMENT
int linearIndex = 2+4*10+5*15*10; // corresponding linear index
func(2,4,5); // specify indices in order of variable id
func(valVec.begin(),valVec.end()); // indices in order of variable id
func(linearIndex); // access by corresponding linear index
func(varVec,valVec); // access by named variables in lists (does not work for arrays)
func(vars,vars+3,valVec.begin(),valVec.end()); // access by named variables using iterators
func(varVec.begin(),varVec.end(),valVec.begin(),valVec.end());
```

Here, the last three variants all work by specifying named variables and corresponding values in lists, either by passing reference to containers directly (which must supply interfaces consistent with standard library containers), or by passing iterators to such lists. In each case, the *k*th value is assumed to correspond to the *k*th named variable. Any values specified for variables that are not in the functions domain are ignored.

For some versions of the `()` operator, the member function [maxsum::DiscreteFunction::at](#) is also provided for convenient element access through pointers. For example, the following code is valid:

```
using namespace maxsum;
registerVariable (3,5);
DiscreteFunction func(3,4.5);
DiscreteFunction* pFunc = &func;
val = (*pFunc)(2); // works, but ugly
val = pFunc->at(2); // equivalent, but more readable
```

1.3.3 Mathematical Operations

Three main types of mathematical operation are provided by the `maxsum::DiscreteFunction` class, each of which are described below.

1.3.3.1 Scalar Operations

Scalar operations are used to calculate specific statistics about a function's values over its domain. The table below summarises the available scalar operations, together with their mathematical definition w.r.t. a `maxsum::DiscreteFunction`, f , with linear indices 1 to N .

Member Function	Definition
<code>DiscreteFunction::mean</code>	$\frac{1}{N} \sum_{k=1}^N f(k)$
<code>DiscreteFunction::max</code>	$\max_k f(k)$
<code>DiscreteFunction::min</code>	$\min_k f(k)$
<code>DiscreteFunction::maxnorm</code>	$\max_k f(k) $

1.3.3.2 Domain Operations

Domain operations operate on a single `DiscreteFunction` object to reduce or expand its domain in some way. In particular, the `DiscreteFunction::expand` member extends the set of variables that a `DiscreteFunction` depends on, such that its new domain is the union of its previous domain, and a set of variables specified by the `expand` function's arguments. Several overloaded versions exist that allow the new set of variables to be specified in different ways:

Member Function	Description
<code>DiscreteFunction::expand(const VarID var)</code>	Expands the domain to include the variable <code>var</code> .
<code>DiscreteFunction::expand(const DiscreteFunction& fun)</code>	Expands the domain of <code>this DiscreteFunction</code> to include all variables in the domain of <code>fun</code> .
<code>template<class VarInd></code> <code>DiscreteFunction::expand(VarInd begin, VarInd end)</code>	Expands the domain to include all variables in the sequence pointed to by the iterators <code>begin</code> and <code>end</code> .

After a call to `expand`, the `DiscreteFunction`'s values remain the same for all elements of its previous domain. For example

```
using namespace maxsum;
registerVariable (1,3);
registerVariable (2,3);
DiscreteFunction func(2,0); // depends only on variable 2
func(0) = 1.1;
func(1) = 2.2;
func(2) = 3.3;

func.expand(1); // func now depends on variables 1 and 2
val = func(1,0); // val == 1.1
val = func(2,0); // val == 1.1
val = func(1,2); // val == 3.3
val = func(2,2); // val == 3.3
```

As well as expanding a `DiscreteFunction`'s domain, it is often necessary to *reduce* its domain. In general, this results in a loss of information, so it is necessary to specify how the values in the new smaller domain are derived from those in the original larger domain. Currently, there are two ways to do this.

First, the `DiscreteFunction::condition` member function can be used to specify fixed values for a set of variables that are to be removed from a function's domain. For example, following on from the code in the last example, we may reduce the domain of `func` as follows.

```
VarID toRemove[] = {2}; // list of variables to remove
ValIndex vals[] = {1}; // fixed values for removed variables
func.condition (toRemove,toRemove+1,vals,vals+1); // specify iterators over arrays
```

```
val = func(0); // val == 2.2 (because variable 2 was given fixed value 1)
val = func(1); // val == 2.2 (see code above)
val = func(2); // val == 2.2
```

Second, rather than conditioning on specific values for the removed variables, we can *marginalise* by somehow aggregating all the values for the removed variables to produce a single value. In particular, the definition of the max-sum algorithm requires variables to be removed by taking a function's maximum value across the removed variables. Here, this is achieved using the [maxMarginal](#) function, which takes the maximum value of one [DiscreteFunction](#) across a set variables, and stores the result in another [DiscreteFunction](#).

This works by passing references to two [DiscreteFunctions](#): the first, `inFun`, is the original function to be marginalised, and the second, `outFun`, is a function with a smaller domain, in which the result will be stored. The domain of `outFun` must not be larger than the domain of `inFun`, and any variables that are in the domain of `inFun`, but not in the domain of `outFun`, will be marginalised. This procedure is demonstrated in the following code.

```
VarID vars = {1,2};
ValIndex siz = {2,3}; // variable 1 has domain size 2, etc.
registerVariables (vars,vars+2,siz,siz+2);
DiscreteFunction inFun(vars,vars+2); // depends on variables 1 and 2
DiscreteFunction outFun(vars,vars+1); // depends on variable 1 only

inFun(0,0) = 1; // assign some values to inFun
inFun(1,1) = 2;
inFun(0,2) = 3;
inFun(1,0) = 4;
inFun(0,1) = 5;
inFun(1,2) = 6;

maxMarginal (inFun,outFun); // max marginalise inFun and store result in outFun

val = outFun(0); // val==5 (maximum value for variable 1=0 in inFun)
val = outFun(1); // val==6 (maximum value for variable 1=1 in inFun)
```

The reason for this style is efficiency: by preallocating `outFun` to store the result, we do not need to allocate temporary objects in memory, and if necessary, can reuse `outFun` to store the result of several similar marginalisations.

More generally, the maxsum library also provides a number marginalisation functions, which aggregate across the removed variables in different ways. These are summarised in the table below. However, only the [maxMarginal](#) function is actually required to implement the max-sum algorithm. See [The MaxSumController Class](#) for details.

Function	Description
maxMarginal	Marginalise by maximising across removed variables.
minMarginal	Marginalise by minimising across removed variables.
meanMarginal	Marginalise by averaging across removed variables.
marginal	Marginalise using a custom aggregation function.

1.3.3.3 Arithmetic Operations

Arithmetic operations on [DiscreteFunction](#)'s are provided by overloading the standard arithmetic operators in C++, including `+`, `-`, and `*`. These operations are applied elementwise across the cartesian product of the operands' domains, as illustrated in the following example:

```
VarID vars = {1,2};
ValIndex siz = {2,3}; // variable 1 has domain size 2, etc.
registerVariables (vars,vars+2,siz,siz+2);
DiscreteFunction f(vars,vars+1); // depends only on variable 1
DiscreteFunction g(vars+1,vars+2); // depends only on variable 2
DiscreteFunction h(vars,vars+2); // depends on variables 1 and 2
```

```
// assign some values
f(0) = 1.0;   f(1) = 1.1;
g(0) = 2.0;   g(1) = 2.1;   g(2) = 2.2;
h(0,0) = 3.0; h(0,1) = 3.1; h(0,2) = 3.2;
h(1,0) = 4.0; h(1,1) = 4.1; h(1,2) = 4.2;

DiscreteFunction x = f+g;
// x(0,0) == f(0) + g(0) == 1.0 + 2.0 == 3.0;
// x(1,0) == f(1) + g(0) == 1.1 + 2.0 == 3.1;
// ....
// x(1,2) == f(1) + g(2) == 1.1 + 2.2 == 3.3;

DiscreteFunction y = f+h;
// y(0,0) == f(0) + h(0,0) == 1.0 + 3.0 == 4.0;
// y(1,0) == f(1) + h(1,0) == 1.1 + 4.0 == 5.1;
// ....
// y(1,2) == f(1) + h(1,2) == 1.1 + 4.2 == 5.3;
```

Here, notice that the domain of the results, x and y , are automatically set to the cartesian product of the domains of the operands. The following operations are all similarly defined.

Operation	Description
$f + g$	Element-wise addition.
$f - g$	Element-wise subtraction.
$f * g$	Element-wise multiplication.
$f += g$	Element-wise addition, storing result in f .
$f -= g$	Element-wise subtraction, storing result in f .
$f *= g$	Element-wise multiplication, storing result in f .
$-g$	Unary Minus.

The right operand, g , can also be replaced by a scalar value of any numeric type convertible to [ValType](#). In this case, the scalar is operated on with each element in the [DiscreteFunction](#)'s domain e.g.

```
DiscreteFunction z = f + 100.5;
// z(0) == f(0) + 100.5 == 1.0 + 100.5 == 101.5
// z(1) == f(1) + 100.5 == 1.1 + 100.5 == 101.6
```

Since adding a number of [DiscreteFunctions](#) together is a common procedure, a member function [DiscreteFunction::add](#) is also provided, which takes iterators to the start and end of a sequence of [DiscreteFunctions](#), and adds them all to the current [DiscreteFunction](#), expanding its domain if necessary.

1.3.4 Comparison Operations

Equality between [DiscreteFunctions](#) can be defined in several ways, and for this reason the [maxsum](#) library provides three different functions for testing equality between [DiscreteFunction](#) objects:

Function	Description
sameDomain(const DiscreteFunction& f1, const DiscreteFunction& f2)	Returns true iff $f1$ and $f2$ have the same domain.
equalWithinTolerance(const DiscreteFunction& f1, const DiscreteFunction& f2, ValType tol)	Returns true iff $f1$ and $f2$ are equal across the cartesian product of their domains, within a given tolerance, tol .
strictlyEqualWithinTolerance(const DiscreteFunction& f1, const DiscreteFunction& f2, ValType tol)	Returns true iff <code>sameDomain(f1, f2) && equalWithinTolerance(f1, f2, tol)</code>

The functions [equalWithinTolerance](#) and [strictlyEqualWithinTolerance](#) are provided so that two [DiscreteFunctions](#) can be treated as equal in cases where their values differ only by some small rounding error. For cases in which strict equality is required between values is required (i.e. the error tolerance is 0), the standard equality and inequality operators may also be used:

Operator	Definition
<code>f1==f2</code>	<code>equalWithinTolerance(f1,f2,0)</code>
<code>f1!=f2</code>	<code>!equalWithinTolerance(f1,f2,0)</code>

1.3.5 Miscellaneous Operations

The [DiscreteFunction](#) class includes a number of members that do not directly relate to a [DiscreteFunction](#)'s values, but instead provide meta data about the function, and enable various house keeping operations. These are summarised in the following table.

Function	Description
DiscreteFunction::domainSize	Returns the size of this DiscreteFunction 's domain, defined as the product of the domain sizes for each variable in its domain.
DiscreteFunction::noVars	Returns the number of variables in this DiscreteFunction 's domain.
DiscreteFunction::dependsOn	Returns <code>true</code> if a specified variable is in this DiscreteFunction 's domain.
DiscreteFunction::varBegin	Returns an iterator to the start of the list of variables in this DiscreteFunction 's domain.
DiscreteFunction::varEnd	Returns an iterator to the end of the list of variables in this DiscreteFunction 's domain.
DiscreteFunction::sizeBegin	Returns an iterator to the start of a list of domain sizes, in which the kth element gives the domain size for the kth variable in this DiscreteFunction 's domain.
DiscreteFunction::sizeEnd	Returns an iterator to the end of a list of domain sizes, in which the kth element gives the domain size for the kth variable in this DiscreteFunction 's domain.

1.4 The DomainIterator Class

Todo.

1.5 The MaxSumController Class

[MaxSumController](#) is the main class responsible for implementing the max-sum algorithm, and providing its results. To run the algorithm using this class, the following steps are all that is required:

1. Register all variables in the factor graph using either [registerVariable](#) or [registerVariables](#).
2. Create a set of [DiscreteFunction](#) objects, representing each factor in the target problem's factor graph.
3. Construct a new [MaxSumController](#) object, optionally specifying termination conditions.
4. Use the [MaxSumController::setFactor](#) member function to specify the set of factors from the previously created list of [DiscreteFunction](#) objects.
5. Call [MaxSumController::optimise](#) to run the max-sum algorithm
6. Use [MaxSumController::valBegin](#) and [MaxSumController::valEnd](#) to iterator through all variables in the factor graph, and retrieve there optimal values.

For example, the following code illustrates how these steps may be implemented in practice:

```
using namespace maxsum;
std::vector<VarID> vars; // list of variables in factor graph
std::vector<ValIndex> sizes; // sizes[k] == domain size for vars[k]
```

```
// ... populate above containers with appropriate values
registerVariables(vars.begin(),vars.end(),sizes.begin(),sizes.end());

std::map<FactorID,DiscreteFunction> factors; // factors mapped to ids
// populate factors with DiscreteFunctions mapped to factor ids

MaxSumController controller(100,0.0001); // new controller with custom stopping conditions
// see MaxSumController::MaxSumController for details
// set up factor graph inside controller
// edges of graph are inferred automatically for factor domains
typedef std::map<FactorID,DiscreteFunction>::const_iterator Iterator;
for(Iterator it=factors.begin(); it!=factors.end(); ++it)
{
    controller.setFactor(it->first,it->second);
}

controller.optimise(); // run the max-sum algorithm

for(MaxSumController::ConstValueIterator it=controller.valBegin();
    it!=controller.valEnd(); ++it)
{
    std::cout << "Optimal value for variable " << it->first
               << " is " << it->second << std::endl;
}
```

In addition to these functions, [MaxSumController](#) class also provides a number of other member functions to set, change, and query the factor graph in different ways. These are particular useful in the problems where the factor values, or the shape of the factor graph can change over time. For further information, see the manual page for [MaxSumController](#).

1.6 Tips on Writing Efficient Code

In general, the guidelines for writing efficient C++ code in any context also apply to code written using the maxsum library. In particular, one key piece of advice is to, as far as possible, avoid creating temporary objects in memory, when existing objects can be modified and reused.

One place where temporary objects are unavoidable is in the use of certain overloaded operators:

- The arithmetic operators, +, −, and * all create temporary objects to store their result
- The postfix increment operator, ++, returns a copy of its operand, before incrementing the operands value.

Where possible, it is therefore better to avoid these operators, and instead use *destructive* alternatives, which replace an operands original value with the result:

Operation	Efficient Alternative
it++	++it
f = f + g	f+=g
f = f − g	f-=g
f = f * g	f*=g
f = − f	f *= -1

When performing arithmetic with DiscreteFunctions, another useful technique to expand a function's domain to its final size, rather than allowing its domain to change incrementally with each operation. For example, suppose DiscreteFunctions *f*, *g* and *h* depend on the single variables 1, 2, and 3 respectively. Now consider the following code.

```
// f starts with domain {1} only
f += g; // f reallocated to expand domain to {1,2}
f += h; // f reallocated to expand domain to {1,2,3}
```

On the otherhand, if we know the final domain in advance, we can improve efficiency by doing the following:

```
VarID vars[] = {1,2,3};  
f.expand(vars,vars+3); // domain of f is now {1,2,3}  
f += g; // more efficient: no domain change necessary  
f += h;
```

1.7 Future Work

Although the basic implementation of this library is now complete, this project is still under going its first phase of active development. Accurate results are thus not yet guaranteed, and in fact, bugs are very much still to be expected.

For now, our main priority is thus continued testing and debugging. Beyond that, we also plan to look at various options for optimising the code. For example, one option is to link to the [eigen3](#) linear algebra library, to take advantage of its hardware vectorisation.

We will also continue to improve this documentation of the API.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

maxsum	Namespace for all public types and functions defined by the Max-Sum library	17
maxsum::util	Utility namespace for types used for maxsum library implementation	28

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

maxsum::BadDomainException	Exception thrown when subindices are incorrectly specified for a maxsum::DiscreteFunction . . .	29
maxsum::util::KeySet< Map >::const_iterator	Iterator type that allows read-only access to the keys of the underlying map	30
maxsum::DiscreteFunction	Class representing functions of sets of variables with discrete domains	30
maxsum::DomainConflictException	Exception thrown when conflicting domains are specified for a variable	41
maxsum::DomainIterator	This class provides methods for iterating over the Cartesian product for a set of variable domains	42
maxsum::EmptyNoticeException	Exception thrown by maxsum::util::PostOffice::popNotice when there are no active notices . .	46
maxsum::InconsistentDomainException	Exception thrown when variable domains are somehow registered as inconsistent	47
maxsum::util::KeySet< Map >	Utility class for presenting the keys of a map in a read-only container	48
maxsum::MaxSumController	This class maintains a factor graph and implements the max-sum algorithm	50
maxsum::NoSuchElementException	Exception thrown when there has been an attempt to access an element of a container that does not exist, and cannot be created on demand	54
maxsum::OutOfRangeException	Exception thrown when indices are out of range	55
maxsum::util::PostOffice< Sender, Receiver, Message >	Class used to store and manage messages sent between factor graph nodes	56
maxsum::util::RefMap< Key, Val >	This class provides a read only wrapper around an existing map, such as that provided by std::map	63
maxsum::UnknownAddressException	Exception thrown when a maxsum::util::PostOffice does not recognise the ID of a Sender or Receiver	65
maxsum::UnknownVariableException	Exception thrown when a variable is referred to, but has not yet been registered using either maxsum::registerVariable or maxsum::registerVariables	66

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

include/ common.h	Common types and functions used by Max-Sum library	67
include/ DiscreteFunction.h	Defines the maxsum::DiscreteFunction class and related utility functions	68
include/ DomainIterator.h	This file defines the maxsum::DomainIterator class	69
include/ exceptions.h	Defines exception classes for all error conditions that may occur in the maxsum library	69
include/ mainpage.h	This header defines the main page for this documentation, and is for that purpose only	70
include/ MaxSumController.h	Defines the MaxSumController class, which implements the max-sum algorithm	71
include/ PostOffice.h	This file defines the utility maxsum::PostOffice class	71
include/ register.h	This Header defines functions for registering the set of all variables on which maxsum::Discrete-Function objects may depend	72
include/ util_containers.h	This file defines some utility container types required for the implementation of the maxsum library	73
src/ DiscreteFunction.cpp	Implementation of member functions of class maxsum::DiscreteFunction	73
src/ DomainIterator.cpp	This file implements the maxsum::DomainIterator class	74
src/ MaxSumController.cpp	Implementation of MaxSumController class members	74
src/ register.cpp	Implementation of functions in register.h	74

Chapter 5

Namespace Documentation

5.1 maxsum Namespace Reference

Namespace for all public types and functions defined by the Max-Sum library.

Namespaces

- namespace [util](#)
Utility namespace for types used for maxsum library implementation.

Classes

- class [DiscreteFunction](#)
Class representing functions of sets of variables with discrete domains.
- class [DomainIterator](#)
This class provides methods for iterating over the Cartesian product for a set of variable domains.
- class [NoSuchElementException](#)
Exception thrown when there has been an attempt to access an element of a container that does not exist, and cannot be created on demand.
- class [EmptyNoticeException](#)
Exception thrown by `maxsum::util::PostOffice::popNotice` when there are no active notices.
- class [UnknownAddressException](#)
Exception thrown when a `maxsum::util::PostOffice` does not recognise the ID of a Sender or Receiver.
- class [BadDomainException](#)
Exception thrown when subindices are incorrectly specified for a `maxsum::DiscreteFunction`.
- class [OutOfRangeException](#)
Exception thrown when indices are out of range.
- class [DomainConflictException](#)
Exception thrown when conflicting domains are specified for a variable.
- class [UnknownVariableException](#)
Exception thrown when a variable is referred to, but has not yet been registered using either `maxsum::registerVariable` or `maxsum::registerVariables`.
- class [InconsistentDomainException](#)
Exception thrown when variable domains are somehow registered as inconsistent.
- class [MaxSumController](#)
This class maintains a factor graph and implements the max-sum algorithm.

Typedefs

- typedef double [ValType](#)
Type of values stored by [maxsum::DiscreteFunction](#) objects.
- typedef unsigned int [VarID](#)
Type used for uniquely identifying variables.
- typedef unsigned int [FactorID](#)
Type used for uniquely identifying factors in a factor graph.
- typedef int [ValIndex](#)
Integer type used for indexing coefficient values.

Functions

- template<class VecType >
void [ind2sub](#) (const VecType &siz, const typename VecType::value_type ind, VecType &sub)
C++ Implementation of Matlab ind2sub function.
- template<class SizIt, class SubIt >
[ValIndex sub2ind](#) (SizIt sizFirst, SizIt sizEnd, SubIt subFirst, SubIt subEnd)
C++ Implementation of Matlab sub2ind function.
- template<class VecType >
VecType::value_type [sub2ind](#) (const VecType &siz, const VecType &sub)
C++ Implementation of Matlab sub2ind function.
- std::ostream & [operator<<](#) (std::ostream &out, const [DiscreteFunction](#) &fun)
Pretty prints a [maxsum::DiscreteFunction](#) Format is similar to the disp function in Matlab for N-D arrays, except that first dimension appears in rows rather than columns.
- bool [sameDomain](#) (const [DiscreteFunction](#) &f1, const [DiscreteFunction](#) &f2)
Check that two [maxsum::DiscreteFunction](#) objects have the same domain.
- bool [equalWithinTolerance](#) (const [DiscreteFunction](#) &f1, const [DiscreteFunction](#) &f2, [ValType](#) tol=DEFAULT_VALUE_TOLERANCE)
Check that two [maxsum::DiscreteFunction](#) objects are equal within a specified tolerance.
- bool [strictlyEqualWithinTolerance](#) (const [DiscreteFunction](#) &f1, const [DiscreteFunction](#) &f2, [ValType](#) tol=DEFAULT_VALUE_TOLERANCE)
Check that two [maxsum::DiscreteFunction](#) objects are equal with a specified tolerance, and have exactly the same domain.
- bool [operator==](#) (const [DiscreteFunction](#) &f1, const [DiscreteFunction](#) &f2)
Return true if functions are equal.
- bool [operator!=](#) (const [DiscreteFunction](#) &f1, const [DiscreteFunction](#) &f2)
Return true if functions are equal.
- template<typename F >
void [marginal](#) (const [DiscreteFunction](#) &inFun, F aggregate, [DiscreteFunction](#) &outFun)
Marginalise a [maxsum::DiscreteFunction](#) using a specified aggregation function.
- void [maxMarginal](#) (const [DiscreteFunction](#) &inFun, [DiscreteFunction](#) &outFun)
Marginalise a [maxsum::DiscreteFunction](#) by maximisation.
- void [minMarginal](#) (const [DiscreteFunction](#) &inFun, [DiscreteFunction](#) &outFun)
Marginalise a [maxsum::DiscreteFunction](#) by minimisation.
- void [meanMarginal](#) (const [DiscreteFunction](#) &inFun, [DiscreteFunction](#) &outFun)
Marginalise a [maxsum::DiscreteFunction](#) by averaging.
- std::ostream & [operator<<](#) (std::ostream &out, [MaxSumController](#) &controller)
Utility function used to dump the current state of this controller for debugging purposes.
- bool [isRegistered](#) ([VarID](#) var)
Returns true if the specified variable is registered.

- `template<class VarIt >`
`bool allRegistered (VarIt varBegin, VarIt varEnd)`
Returns true if all specified variables are registered.
- `ValIndex getDomainSize (VarID var)`
Returns the registered domain size for a specified variable.
- `int getNumOfRegisteredVariables ()`
Returns the number of currently registered variables.
- `void registerVariable (VarID var, ValIndex siz)`
Registers a variable with a specified domain size.
- `template<class VarIt, class IndIt >`
`void registerVariables (VarIt varBegin, VarIt varEnd, IndIt sizBegin, IndIt sizEnd)`
Register a list of variables with specified domain sizes.

Variables

- `const ValType DEFAULT_VALUE_TOLERANCE = DBL_EPSILON * 1000.0`
Default tolerance used for comparing values of type `maxsum::ValType`.

5.1.1 Detailed Description

Namespace for all public types and functions defined by the Max-Sum library.

5.1.2 Typedef Documentation

5.1.2.1 `typedef unsigned int maxsum::FactorID`

Type used for uniquely identifying factors in a factor graph.

This is purposely an integer type, because we want this to be efficient for storing and passing between functions. Note: bitwise operations can be used in some addressing schemes, e.g. something like IP addresses.

See Also

[maxsum::VarID](#)

5.1.2.2 `typedef int maxsum::ValIndex`

Integer type used for indexing coefficient values.

This is the value type for all variables that are referenced and identified using `maxsum::VarID`. In particular, if a value of type `maxsum::ValIndex` is specified for each variable in a `maxsum::DiscreteFunction` object's domain, then exactly one value of type `maxsum::ValType` will be returned by element accessor functions, such as `maxsum::DiscreteFunction::at`

5.1.2.3 `typedef double maxsum::ValType`

Type of values stored by `maxsum::DiscreteFunction` objects.

This is, this type is used to represent the codomain of mathematical functions represented by `maxsum::DiscreteFunction` objects.

See Also

[maxsum::DiscreteFunction](#)

5.1.2.4 typedef unsigned int maxsum::VarID

Type used for uniquely identifying variables.

This is purposely an integer type, because we want this to be efficient for storing and passing between functions.
Note: bitwise operations can be used in some addressing schemes, e.g. something like IP addresses.

See Also

[maxsum::FactorID](#)

5.1.3 Function Documentation

5.1.3.1 template<class VarIt > bool maxsum::allRegistered (VarIt varBegin, VarIt varEnd)

Returns true if all specified variables are registered.

Parameters are iterators over a list of variable ids of type [maxsum::VarID](#).

Parameters

<i>varBegin</i>	iterator to beginning of variable list.
<i>varEnd</i>	iterator to end of variable list.

Returns

true if all registered, false otherwise.

5.1.3.2 bool maxsum::equalWithinTolerance (const DiscreteFunction & f1, const DiscreteFunction & f2, ValType tol = DEFAULT_VALUE_TOLERANCE)

Check that two [maxsum::DiscreteFunction](#) objects are equal within a specified tolerance.

This function returns true if and only if, for all k :

$$-tol < 1 - f1(k) / f2(k) < tol$$

Parameters

in	<i>f1</i>	First function to compare
in	<i>f2</i>	Second function to compare
in	<i>tol</i>	tolerance used for comparing values

See Also

[maxsum::DEFAULT_VALUE_TOLERANCE](#)

5.1.3.3 maxsum::ValIndex maxsum::getDomainSize (VarID var)

Returns the registered domain size for a specified variable.

Returns the domain size for a specified variable.

Parameters

<i>var</i>	id of the variable to search for.
------------	-----------------------------------

Returns

domain size of var

Exceptions

<i>UnknownVariableException</i>	if the variable is not registered.
---	------------------------------------

Parameters

<i>var</i>	id of the variable to search for.
------------	-----------------------------------

Returns

domain size of var

Exceptions

<i>UnknownVariableException</i>	if variable is not registered.
---	--------------------------------

5.1.3.4 int maxsum::getNumOfRegisteredVariables ()

Returns the number of currently registered variables.

Returns

the number of currently registered variables.

5.1.3.5 template<class VecType > void maxsum::ind2sub (const VecType & siz, const typename VecType::value_type ind, VecType & sub)

C++ Implementation of Matlab ind2sub function.

Main difference here is that indices start from 0. In the special case where siz is empty, ind will also be empty.

Parameters

<i>in</i>	<i>siz</i>	of N-D array
<i>in</i>	<i>ind</i>	linear index
<i>out</i>	<i>sub</i>	vector in which we will put the sub indices.

Postcondition

previous contents of sub will be overwritten.

5.1.3.6 bool maxsum::isRegistered (VarID var)

Returns true if the specified variable is registered.

Parameters

<i>var</i>	id of the variable to search for.
------------	-----------------------------------

Returns

true if the specified variable is registered.
 true if the specified variable is registered.

5.1.3.7 `template<typename F > void maxsum::marginal (const DiscreteFunction & inFun, F aggregate, DiscreteFunction & outFun)`

Marginalise a [maxsum::DiscreteFunction](#) using a specified aggregation function.

This function reduces the domain of `inFun` to that of `outFun` and stores the result in `outFun`. This is done by aggregating over all variables that are in the domain of `inFun`, but not in the domain of `outFun`.

Aggregation is performed by a functor or function pointer, `aggregate`, which is passed as a template parameter. `aggregate` should be defined such that, in the code below, `z` is the aggregation of the function values `x` and `y`.

```
ValType x = inFun(k);
ValType y = inFun(t);
ValType z = aggregate(x,y); // result = aggregate(prevResult,nextVal)
```

Note that `aggregate` may be a function pointer, or any other object for which the above syntax is valid, such as a class instance that overloads the `()` operator, which may depend some internal state. The first argument to `aggregate` should always be the previous estimate, while the second is always the next value to be aggregated.

Various specialisations of this function are provided by [maxsum::maxMarginal\(\)](#), [maxsum::minMarginal\(\)](#), [maxsum::meanMarginal\(\)](#).

In the special case where the domain of `outFun` is equal to the domain of `inFun`, `outFun` becomes a copy of `inFun`

Precondition

`aggregate` is a functor with signature `ValType aggregate(ValType prevResult, ValType nextVal)`
 variables in domain of `outFun` are a subset of variables in `inFun`.

Postcondition

previous content of `outFun` is overwritten.
 The domains of `outFun` and `inFun` remain unchanged.

Template Parameters

<i>F</i>	type of parameter <code>aggregate</code> .
----------	--

Parameters

<code>in</code>	<i>inFun</i>	function to marginalise
<code>in</code>	<i>aggregate</i>	functor or function pointer used to aggregate results.
<code>out</code>	<i>outFun</i>	maxsum::DiscreteFunction in which to store result.

Exceptions

maxsum::BadDomain-Exception	is the domain of <code>outFun</code> is not a subset of <code>inFun</code> .
---	--

See Also

[maxsum::maxMarginal\(\)](#)
[maxsum::minMarginal\(\)](#)
[maxsum::meanMarginal\(\)](#)

5.1.3.8 void maxsum::maxMarginal (const DiscreteFunction & *inFun*, DiscreteFunction & *outFun*)

Marginalise a [maxsum::DiscreteFunction](#) by maximisation.

This function reduces the domain of *inFun* to that of *outFun* by maximisation, and stores the result in *outFun*. This behaviour is equivalent to `maxsum::marginal(inFun,std::max<ValType>,outFun)`.

Precondition

variables in domain of *outFun* are a subset of variables in *inFun*.

Postcondition

previous content of *outFun* is overwritten.
 The domains of *outFun* and *inFun* remain unchanged.

Parameters

in	<i>inFun</i>	function to marginalise
out	<i>outFun</i>	maxsum::DiscreteFunction in which to store result.

Exceptions

maxsum::BadDomain-Exception	is the domain of <i>outFun</i> is not a subset of <i>inFun</i> .
---	--

See Also

[maxsum::marginal\(\)](#)
[maxsum::minMarginal\(\)](#)
[maxsum::meanMarginal\(\)](#)

5.1.3.9 void maxsum::meanMarginal (const DiscreteFunction & *inFun*, DiscreteFunction & *outFun*)

Marginalise a [maxsum::DiscreteFunction](#) by averaging.

This function reduces the domain of *inFun* to that of *outFun* by averaging, and stores the result in *outFun*.

Precondition

variables in domain of *outFun* are a subset of variables in *inFun*.

Postcondition

previous content of *outFun* is overwritten.
 The domains of *outFun* and *inFun* remain unchanged.

Parameters

in	<i>inFun</i>	function to marginalise
out	<i>outFun</i>	maxsum::DiscreteFunction in which to store result.

Exceptions

<i>maxsum::BadDomain-Exception</i>	is the domain of outFun is not a subset of inFun.
--	---

See Also

[maxsum::marginal\(\)](#)
[maxsum::minMarginal\(\)](#)
[maxsum::maxMarginal\(\)](#)

5.1.3.10 void maxsum::minMarginal (const DiscreteFunction & inFun, DiscreteFunction & outFun)

Marginalise a [maxsum::DiscreteFunction](#) by minimisation.

Marginal a [maxsum::DiscreteFunction](#) by minimisation.

This function reduces the domain of inFun to that of outFun by minimisation, and stores the result in outFun. This behaviour is equivalent to `maxsum::marginal(inFun,std::min<ValType>,outFun)`.

Precondition

variables in domain of outFun are a subset of variables in inFun.

Postcondition

previous content of outFun is overwritten.
The domains of outFun and inFun remain unchanged.

Parameters

in	<i>inFun</i>	function to marginalise
out	<i>outFun</i>	maxsum::DiscreteFunction in which to store result.

Exceptions

<i>maxsum::BadDomain-Exception</i>	is the domain of outFun is not a subset of inFun.
--	---

See Also

[maxsum::marginal\(\)](#)
[maxsum::maxMarginal\(\)](#)
[maxsum::meanMarginal\(\)](#)

This function reduces the domain of inFun to that of outFun by minimisation, and stores the result in outFun. This behaviour is equivalent to `maxsum::marginal(inFun,std::min<ValType>,outFun)`.

Precondition

variables in domain of outFun are a subset of variables in inFun.

Postcondition

previous content of outFun is overwritten.
The domains of outFun and inFun remain unchanged.

Parameters

<code>in</code>	<code>inFun</code>	function to marginalise
-----------------	--------------------	-------------------------

Exceptions

<code>maxsum::BadDomain-Exception</code>	is the domain of outFun is not a subset of inFun.
--	---

Parameters

<code>out</code>	<code>outFun</code>	<code>maxsum::DiscreteFunction</code> in which to store result.
------------------	---------------------	---

See Also

[`maxsum::marginal\(\)`](#)
[`maxsum::maxMarginal\(\)`](#)
[`maxsum::meanMarginal\(\)`](#)

5.1.3.11 `bool maxsum::operator!= (const DiscreteFunction & f1, const DiscreteFunction & f2)` `[inline]`

Return true if functions are equal.

Two functions are equal if they have the same value over the cartesian product of their domains.

5.1.3.12 `std::ostream & maxsum::operator<< (std::ostream & out, const DiscreteFunction & fun)`

Pretty prints a [`maxsum::DiscreteFunction`](#) Format is similar to the disp function in Matlab for N-D arrays, except that first dimension appears in rows rather than columns.

Pretty prints a [`maxsum::DiscreteFunction`](#).

Pretty prints this function.

Format is similar to the disp function in Matlab for N-D arrays, except that first dimension appears in rows rather than columns.

5.1.3.13 `bool maxsum::operator==(const DiscreteFunction & f1, const DiscreteFunction & f2)` `[inline]`

Return true if functions are equal.

Two functions are equal if they have the same value over the cartesian product of their domains. This

5.1.3.14 `void maxsum::registerVariable (VarID var, ValIndex siz)`

Registers a variable with a specified domain size.

Puts the specified variable in a global register, and stores its domain size. Variables can be registered multiple times, but their domain size must never change.

Exceptions

<code>InconsistentDomain-Exception</code>	if this variable is already registered, but with a different domain size.
---	---

Parameters

<i>var</i>	the unique id of this variable
<i>siz</i>	the domain size of this variable

See Also

[maxsum::registerVariables](#)

Put the specified variable in a global register, and stores its domain size. Variables can be registered multiple times, but their domain size must never change.

Exceptions

<i>InconsistentDomain-Exception</i>	if this variable is already registered, but with a different domain size.
---	---

Parameters

<i>var</i>	the unique id of this variable
<i>siz</i>	the domain size of this variable

5.1.3.15 `template<class VarIt , class IndIt > void maxsum::registerVariables (VarIt varBegin, VarIt varEnd, IndIt sizBegin, IndIt sizEnd)`

Register a list of variables with specified domain sizes.

This works in the same way as [maxsum::registerVariable](#) - but does so for multiple variables at a time. The parameters `varBegin` and `varEnd` are iterators to the beginning and end of a list of variable ids, while `sizBegin` and `sizEnd` specify the start and end of a list of their respective domain sizes. Both lists must be ordered such that the *k*th element of the size list is the domain size for the *k*th variable in the variable list.

Parameters

<i>varBegin</i>	iterator to the start of a list of maxsum::VarID
<i>varEnd</i>	iterator to the end of a list of maxsum::VarID
<i>sizBegin</i>	iterator to the start of a list of maxsum::ValIndex
<i>sizEnd</i>	iterator to the end of a list of maxsum::ValIndex

See Also

[maxsum::registerVariable](#)

5.1.3.16 `bool maxsum::sameDomain (const DiscreteFunction & f1, const DiscreteFunction & f2)`

Check that two [maxsum::DiscreteFunction](#) objects have the same domain.

Two functions have the same domain, if they depend on the same set of variables.

Parameters

<i>in</i>	<i>f1</i>	First function to compare
<i>in</i>	<i>f2</i>	Second functions to compare

Returns

true if both function have the same domain.

5.1.3.17 `bool maxsum::strictlyEqualWithinTolerance (const DiscreteFunction & f1, const DiscreteFunction & f2, ValType tol = DEFAULT_VALUE_TOLERANCE)`

Check that two [maxsum::DiscreteFunction](#) objects are equal with a specified tolerance, and have exactly the same domain.

This function returns true if and only if:

```
true == sameDomain(f1, f2) && equalWithinTolerance(f1, f2, tol)
```

Parameters

in	<i>f1</i>	First function to compare
in	<i>f2</i>	Second function to compare
in	<i>tol</i>	tolerance used for comparing values

See Also

[maxsum::DEFAULT_VALUE_TOLERANCE](#)

5.1.3.18 `template<class SizIt, class SubIt> ValIndex maxsum::sub2ind (SizIt sizFirst, SizIt sizEnd, SubIt subFirst, SubIt subEnd)`

C++ Implementation of Matlab sub2ind function.

Main difference here is that indices start from 0.

Parameters

in	<i>sizFirst</i>	iterator to first element of size array.
in	<i>sizEnd</i>	iterator to end element of size array.
in	<i>subFirst</i>	iterator to first element of subindex array.
in	<i>subEnd</i>	iterator to first element of subindex array.

Returns

linear index

5.1.3.19 `template<class VecType> VecType::value_type maxsum::sub2ind (const VecType & siz, const VecType & sub)`

C++ Implementation of Matlab sub2ind function.

Main difference here is that indices start from 0.

Parameters

in	<i>siz</i>	of N-D array
in	<i>sub</i>	empty vector in which we will put the sub indices.

Returns

linear index

5.1.4 Variable Documentation

5.1.4.1 `const ValType maxsum::DEFAULT_VALUE_TOLERANCE = DBL_EPSILON * 1000.0`

Default tolerance used for comparing values of type `maxsum::ValType`.

This is the default value used by the `maxsum::equalWithinTolerance` function, when comparing to `maxsum::DiscreteFunction` objects for equality. Note, if `ValType` is ever redefined, then this value should be changed appropriately also.

See Also

[maxsum::equalWithinTolerance](#)

5.2 `maxsum::util` Namespace Reference

Utility namespace for types used for maxsum library implementation.

Classes

- class [PostOffice](#)
Class used to store and manage messages sent between factor graph nodes.
- class [KeySet](#)
Utility class for presenting the keys of a map in a read-only container.
- class [RefMap](#)
This class provides a read only wrapper around an existing map, such as that provided by `std::map`.

Typedefs

- typedef [PostOffice](#)< [VarID](#),
[FactorID](#) > [V2FPostOffice](#)
Convenience typedef for Variable to Factor PostOffices.
- typedef [PostOffice](#)< [FactorID](#),
[VarID](#) > [F2VPostOffice](#)
Convenience typedef for Factor to Variable PostOffices.
- typedef [PostOffice](#)< [FactorID](#),
[FactorID](#) > [F2FPostOffice](#)
Convenience typedef for Factor to Factor PostOffices.

5.2.1 Detailed Description

Utility namespace for types used for maxsum library implementation. The contents of this namespace are not intended to form part of the external interface to the maxsum library, and can be safely ignored by third party developers.

Chapter 6

Class Documentation

6.1 maxsum::BadDomainException Class Reference

Exception thrown when subindices are incorrectly specified for a [maxsum::DiscreteFunction](#).

```
#include <exceptions.h>
```

Public Member Functions

- [BadDomainException](#) (const std::string where_, const std::string mesg_) throw ()
Constructs a new exception with specified location and message.
- const char * [what](#) () const throw ()
Returns a message describing the cause of this exception, and the location it was thrown.
- virtual ~[BadDomainException](#) () throw ()
Destroys this exception and free's its allocated resources.

Protected Attributes

- const std::string [where](#)
String identifying the source code locatioin where this exceptioin was generated.
- const std::string [mesg](#)
Message describing the cause of this exception.

6.1.1 Detailed Description

Exception thrown when subindices are incorrectly specified for a [maxsum::DiscreteFunction](#).

6.1.2 Constructor & Destructor Documentation

6.1.2.1 maxsum::BadDomainException::BadDomainException (const std::string where_, const std::string mesg_) throw ()
[inline]

Constructs a new exception with specified location and message.

Parameters

in	<i>where_</i>	the source code location where this exception was generated.
in	<i>mesg_</i>	Message describing the reason for this exception.

6.1.3 Member Function Documentation

6.1.3.1 `const char* maxsum::BadDomainException::what () const throw ()` `[inline]`

Returns a message describing the cause of this exception, and the location it was thrown.

Returns

a message describing the cause of this exception, and the location it was thrown.

The documentation for this class was generated from the following file:

- `include/exceptions.h`

6.2 `maxsum::util::KeySet< Map >::const_iterator` Class Reference

Iterator type that allows read-only access to the keys of the underlying map.

```
#include <util_containers.h>
```

Public Member Functions

- `const_iterator` (typename `Map::const_iterator` it)
Constructs a new `KeySet` iterator from a map iterator.
- `const KeySet::value_type & operator* () const`
Overrides dereference operator so that a direct reference to the Map key is returned.

6.2.1 Detailed Description

```
template<class Map>class maxsum::util::KeySet< Map >::const_iterator
```

Iterator type that allows read-only access to the keys of the underlying map.

This is achieved by overriding the map iterator's own dereferencing operator, so that only a constant reference to the key is returned.

The documentation for this class was generated from the following file:

- `include/util_containers.h`

6.3 `maxsum::DiscreteFunction` Class Reference

Class representing functions of sets of variables with discrete domains.

```
#include <DiscreteFunction.h>
```

Public Types

- `typedef std::vector< VarID > ::const_iterator VarIterator`
Type of iterator returned by `DiscreteFunction::varBegin()` and `DiscreteFunction::varEnd()` functions.
- `typedef std::vector< ValIndex > ::const_iterator SizeIterator`
Type of iterator returned by `DiscreteFunction::sizeBegin()` and `DiscreteFunction::sizeEnd()` functions.

Public Member Functions

- [DiscreteFunction](#) ([ValType](#) val=0)
Default Constructor creates constant function that depends on no variables.
- `template<class VarIt >`
[DiscreteFunction](#) (VarIt begin, VarIt end, [ValType](#) val=0)
Constructs function depending on specified variables.
- [DiscreteFunction](#) ([VarID](#) var, [ValType](#) val)
Constructs a function that depends on only one variable.
- [DiscreteFunction](#) (const [DiscreteFunction](#) &val)
Copy Constructor performs deep copy.
- [ValIndex](#) domainSize () const
Accessor method for the total size this function's domain.
- `bool` [dependsOn](#) ([VarID](#) var) const
Returns true if this function depends on the specified variable.
- [VarIterator](#) varBegin () const
Returns an iterator to the start of this function's domain variable list.
- [VarIterator](#) varEnd () const
Returns an iterator to the end of this function's domain variable list.
- [SizeIterator](#) sizeBegin () const
Returns an iterator to the start of this function's domain variable size list.
- [SizeIterator](#) sizeEnd () const
Returns an iterator to the end of this function's domain variable size list.
- `int` noVars () const
Returns the number of variables on which this function depends.
- [DiscreteFunction](#) & operator= ([ValType](#) val)
Sets this function to a constant scalar value.
- [DiscreteFunction](#) & operator= (const [DiscreteFunction](#) &val)
Sets this function to be equal to another.
- [DiscreteFunction](#) & operator+= ([ValType](#) val)
Adds a scalar value to this function.
- [DiscreteFunction](#) & operator-= ([ValType](#) val)
Subtracts a scalar value from this function.
- [DiscreteFunction](#) & operator*= ([ValType](#) val)
Multiplies this function by a scalar.
- [DiscreteFunction](#) operator- ()
Multiply function by -1.
- [DiscreteFunction](#) & operator+= (const [DiscreteFunction](#) &rhs)
Adds a function to this one, expanding the domain if necessary.
- [DiscreteFunction](#) & operator-= (const [DiscreteFunction](#) &rhs)
Subtracts a function from this one, expanding domain if necessary.
- [DiscreteFunction](#) & operator*= (const [DiscreteFunction](#) &rhs)
Multiplies this function by another, expanding domain if necessary.
- `template<class T >`
[DiscreteFunction](#) operator- (T rhs) const
Subtract function or scalar.
- `template<class T >`
[DiscreteFunction](#) operator+ (T rhs) const
Add function or scalar.
- `template<class T >`
[DiscreteFunction](#) operator* (T rhs) const

- Multiply function or scalar.*

 - `template<class VecIt >`
`DiscreteFunction & add (VecIt begin, VecIt end)`
Adds a list of Functions to this one, expanding the domain if necessary.
 - `ValType & operator() (ValIndex ind)`
Access coefficient using linear index.
 - `const ValType & operator() (ValIndex ind) const`
Access coefficient using linear index.
 - `ValType & operator() (ValIndex ind1, ValIndex ind2,...)`
Access coefficient using subindices specified in argument list.
 - `const ValType & operator() (ValIndex ind1, ValIndex ind2,...) const`
Access coefficient using subindices specified in argument list.
 - `ValType & at (ValIndex ind)`
Access coefficient using scalar index.
 - `const ValType & at (ValIndex ind) const`
Access coefficient using scalar index.
 - `ValType & at (ValIndex ind1, ValIndex ind2,...)`
Access coefficient using subindices specified as arguments This is equivalent to DiscreteFunction::operator() but is more convenient when for use with pointers e.g.
 - `const ValType & at (ValIndex ind1, ValIndex ind2,...) const`
Access coefficient using subindices specified as arguments This is equivalent to DiscreteFunction::operator() but is more convenient when for use with pointers e.g.
 - `template<class IndIt >`
`ValType & operator() (IndIt begin, IndIt end)`
Access coefficient by subindices.
 - `template<class IndIt >`
`const ValType & operator() (IndIt begin, IndIt end) const`
Access coefficient by subindices.
 - `template<class VarIt , class IndIt >`
`ValType & operator() (VarIt varBegin, VarIt varEnd, IndIt indBegin, IndIt indEnd)`
Access coefficient by subindices for specified variables.
 - `template<class VarIt , class IndIt >`
`const ValType & operator() (VarIt varBegin, VarIt varEnd, IndIt indBegin, IndIt indEnd) const`
Access coefficient by subindices for specified variables.
 - `template<class VarVec , class IndVec >`
`ValType & operator() (VarVec var, IndVec ind)`
Access coefficient by subindices for specified variables.
 - `template<class VarVec , class IndVec >`
`const ValType & operator() (VarVec var, IndVec ind) const`
Access coefficient by subindices for specified variables.
 - `ValType & operator() (const DomainIterator &it)`
Access coefficient by subindices specified by the current indices specified by a `maxsum::DomainIterator`.
 - `const ValType & operator() (const DomainIterator &it) const`
Access coefficient by subindices specified by the current indices specified by a `maxsum::DomainIterator`.
 - `void swap (DiscreteFunction &fun)`
Swaps the value and domain of this function with another.
 - `template<class VarInd >`
`void expand (VarInd begin, VarInd end)`
Make this function depend on additional variables.
 - `void expand (const VarID var)`
Expand the domain of this function to include a named variable.
 - `void expand (const DiscreteFunction &fun)`

Make the domain of this function include the domain of another.

- `template<class VarIt, class IndIt >`
`void condition (VarIt vBegin, VarIt vEnd, IndIt iBegin, IndIt iEnd)`
Condition function on specified variable values.
- `ValType min () const`
Returns the minimum scalar value of the function across entire domain.
- `ValType max () const`
Returns the maximum scalar value for function across entire domain.
- `ValIndex argmax () const`
Returns the linear index of the maximum value accross entire domain.
- `ValType maxnorm () const`
Returns the maxnorm for this function.
- `ValType mean () const`
Returns the mean scalar value for function across entire domain.

Friends

- `std::ostream & operator<< (std::ostream &out, const DiscreteFunction &fun)`
Pretty prints this function.

6.3.1 Detailed Description

Class representing functions of sets of variables with discrete domains.

Template Parameters

<i>ValType</i>	the scalar type of value returned by this function. We expect this to be a primitive numeric type, such as double or int.
----------------	---

6.3.2 Constructor & Destructor Documentation

6.3.2.1 maxsum::DiscreteFunction::DiscreteFunction (ValType val = 0) [inline]

Default Constructor creates constant function that depends on no variables.

Parameters

<i>in</i>	<i>val</i>	the constant scalar value of this function.
-----------	------------	---

6.3.2.2 template<class VarIt > maxsum::DiscreteFunction::DiscreteFunction (VarIt begin, VarIt end, ValType val = 0) [inline]

Constructs function depending on specified variables.

When passing variable ids, we assume that the list is unique and sorted. Initially, for any input the function returns the value of the val parameter, which defaults to zero.

Parameters

<i>in</i>	<i>begin</i>	iterator to start of variable list.
<i>in</i>	<i>end</i>	iterator to end of variable list.
<i>in</i>	<i>val</i>	initial scalar value output for function.

Exceptions

UnknownVariableException	if any variable in the list specified by <code>begin</code> and <code>end</code> is not registered.
--	---

6.3.2.3 `maxsum::DiscreteFunction::DiscreteFunction (VarID var, ValType val)` `[inline]`

Constructs a function that depends on only one variable.

Parameters

<code>in</code>	<code>var</code>	the variable to put in this function's domain.
<code>in</code>	<code>val</code>	scalar value used to initialise function across domain.

Postcondition

$$\forall k \ f(k) = val$$

Exceptions

UnknownVariableException	if <code>var</code> is not registered.
--	--

6.3.2.4 `maxsum::DiscreteFunction::DiscreteFunction (const DiscreteFunction & val)` `[inline]`

Copy Constructor performs deep copy.

Parameters

<code>in</code>	<code>val</code>	the object to copy.
-----------------	------------------	---------------------

6.3.3 Member Function Documentation

6.3.3.1 `template<class VecIt > DiscreteFunction& maxsum::DiscreteFunction::add (VecIt begin, VecIt end)` `[inline]`

Adds a list of Functions to this one, expanding the domain if necessary.

The result is $this + \sum_k func[k]$

Returns

reference to this function.

6.3.3.2 `ValType& maxsum::DiscreteFunction::at (ValIndex ind)` `[inline]`

Access coefficient using scalar index.

This is equivalent to `DiscreteFunction::operator()(maxsum::ValIndex ind)` but is more convenient when for use with pointers e.g.

```
x->(k); // illegal
x->at(k); // ok
```

See Also

[DiscreteFunction::operator\(\)\(maxsum::ValIndex ind\)](#)

6.3.3.3 `const ValType& maxsum::DiscreteFunction::at (ValIndex ind) const` `[inline]`

Access coefficient using scalar index.

This is equivalent to `DiscreteFunction::operator()(maxsum::ValIndex ind) const` but is more convenient when for use with pointers e.g.

```
x->(k); // illegal
x->at(k); // ok
```

See Also

[DiscreteFunction::operator\(\)\(maxsum::ValIndex ind\) const](#)

6.3.3.4 `ValType & DiscreteFunction::at (ValIndex ind1, ValIndex ind2, ...)`

Access coefficient using subindices specified as arguments This is equivalent to `DiscreteFunction::operator()` but is more convenient when for use with pointers e.g.

```
x->(k); // illegal
x->at(k); // ok
```

See Also

`DiscreteFunction::operator()`

```
x->(k); // illegal
x->at(k); // ok
```

See Also

`DiscreteFunction::operator()`

6.3.3.5 `const ValType & DiscreteFunction::at (ValIndex ind1, ValIndex ind2, ...) const`

Access coefficient using subindices specified as arguments This is equivalent to `DiscreteFunction::operator()` but is more convenient when for use with pointers e.g.

```
x->(k); // illegal
x->at(k); // ok
```

See Also

`DiscreteFunction::operator()`

```
x->(k); // illegal
x->at(k); // ok
```

See Also

`DiscreteFunction::operator()`

6.3.3.6 `template<class VarIt , class IndIt > void maxsum::DiscreteFunction::condition (VarIt vBegin, VarIt vEnd, IndIt iBegin, IndIt iEnd) [inline]`

Condition function on specified variable values.

Changes this function so that it does not depend on any of the variables in the list specified by varBegin and varEnd, by conditioning this variables on a corresponding list of values.

Parameters

in	<i>vBegin</i>	iterator to start of variable list.
in	<i>vEnd</i>	iterator to end of variable list.
in	<i>iBegin</i>	iterator to start of value list.
in	<i>iEnd</i>	iterator to end of value list.

Precondition

parameters must be iterators over *sorted* lists.

Postcondition

After calling this method, this [DiscreteFunction](#) will not depend on any of the variables in the list specified by varBegin and varEnd.

6.3.3.7 `bool DiscreteFunction::dependsOn (VarID var) const`

Returns true if this function depends on the specified variable.

Parameters

in	<i>var</i>	The id of the variable to search for in this function's domain.
----	------------	---

Returns

true if *var* is in this function's domain.

6.3.3.8 `template<class VarInd > void maxsum::DiscreteFunction::expand (VarInd begin, VarInd end) [inline]`

Make this function depend on additional variables.

If necessary, the domain of this function is expanded to include the specified list of variables.

Parameters

in	<i>begin</i>	iterator to first variable to add
in	<i>end</i>	iterator to end of list of variables.

6.3.3.9 `void DiscreteFunction::expand (const VarID var)`

Expand the domain of this function to include a named variable.

Parameters

in	<i>var</i>	the id of the variable to add to this function's domain.
----	------------	--

Postcondition

The domain of this function is the union of its previous domain, and the additional specified variable.

6.3.3.10 void DiscreteFunction::expand (const DiscreteFunction & fun)

Make the domain of this function include the domain of another.

If necessary, the domain of this function is expanded to include the domain of the parameter fun.

Parameters

<i>in</i>	<i>fun</i>	function whose domain we want to expand to.
-----------	------------	---

Postcondition

domain of this is union of its previous domain, with that of fun.

If necessary, the domain of this function is expanded to include the domain of the parameter fun.

Parameters

<i>in</i>	<i>fun</i>	function whose domain we want to expand to.
-----------	------------	---

Postcondition

domain of this is union of its previous domain, with that of fun.

6.3.3.11 ValType DiscreteFunction::maxnorm () const

Returns the maxnorm for this function.

The maxnorm is defined as the maximum absolute value of the function:

$$\|f(k)\|_{\infty} = \max_k |f(k)|$$

6.3.3.12 ValType DiscreteFunction::min () const

Returns the minimum scalar value of the function across entire domain.

Returns the maximum scalar value for function across entire domain.

6.3.3.13 ValType & DiscreteFunction::operator() (ValIndex ind1, ValIndex ind2, ...)

Access coefficient using subindices specified in argument list.

Number of arguments needs to match the number of variables in this function's domain, otherwise bad things will happen.

6.3.3.14 template<class VarIt, class IndIt> ValType& maxsum::DiscreteFunction::operator() (VarIt varBegin, VarIt varEnd, IndIt indBegin, IndIt indEnd) [inline]

Access coefficient by subindices for specified variables.

Specified variables must be superset of variables on which this function depends.

6.3.3.15 `template<class VarIt , class IndIt > const ValType& maxsum::DiscreteFunction::operator() (VarIt varBegin, VarIt varEnd, IndIt indBegin, IndIt indEnd) const` `[inline]`

Access coefficient by subindices for specified variables.

Specified variables must be superset of variables on which this function depends.

6.3.3.16 `template<class VarVec , class IndVec > ValType& maxsum::DiscreteFunction::operator() (VarVec var, IndVec ind)` `[inline]`

Access coefficient by subindices for specified variables.

Specified variables must be superset of variables on which this function depends.

6.3.3.17 `template<class VarVec , class IndVec > const ValType& maxsum::DiscreteFunction::operator() (VarVec var, IndVec ind) const` `[inline]`

Access coefficient by subindices for specified variables.

Specified variables must be superset of variables on which this function depends.

6.3.3.18 `ValType & DiscreteFunction::operator() (const DomainIterator & it)`

Access coefficient by subindices specified by the current indices specified by a [maxsum::DomainIterator](#).

Note: this works by accessing [maxsum::DomainIterator::getSubInd\(\)](#) array. If the domain of the [maxsum::DomainIterator](#) exactly matches the domain of this function, then it is more efficient to index using the corresponding linear index instead. That is

```
maxsum::DiscreteFunction f(...); // some function
maxsum::DomainIterator it(f); // iterator over domain of f
maxsum::ValType x = f(it); // this works
maxsum::ValType x = f(it.getInd()); // more efficient in this case.
```

Parameters

<code>in</code>	<code>it</code>	Iterator used to index this function.
-----------------	-----------------	---------------------------------------

See Also

[maxsum::DomainIterator::getSubInd\(\)](#)
[maxsum::DomainIterator::getInd\(\)](#)

Note: this works by accessing [maxsum::DomainIterator::getSubInd\(\)](#) array. If the domain of the [maxsum::DomainIterator](#) exactly matches the domain of this function, then it is more efficient to index using the corresponding linear index instead. That is

```
maxsum::DiscreteFunction f(...); // some function
maxsum::DomainIterator it(f); // iterator over domain of f
maxsum::ValType x = f(it); // this works
maxsum::ValType x = f(it.getInd()); // more efficient in this case.
```

Parameters

<code>in</code>	<code>it</code>	Iterator used to index this function.
-----------------	-----------------	---------------------------------------

See Also

[maxsum::DomainIterator::getSubInd\(\)](#)
[maxsum::DomainIterator::getInd\(\)](#)

6.3.3.19 `const ValType & DiscreteFunction::operator()(const DomainIterator & it) const`

Access coefficient by subindices specified by the current indices specified by a [maxsum::DomainIterator](#).

Note: this works by accessing [maxsum::DomainIterator::getSubInd\(\)](#) array. If the domain of the [maxsum::DomainIterator](#) exactly matches the domain of this function, then it is more efficient to index using the corresponding linear index instead. That is

```
maxsum::DiscreteFunction f(...); // some function
maxsum::DomainIterator it(f); // iterator over domain of f
maxsum::ValType x = f(it); // this works
maxsum::ValType x = f(it.getInd()); // more efficient in this case.
```

Parameters

<code>in</code>	<code>it</code>	Iterator used to index this function.
-----------------	-----------------	---------------------------------------

See Also

[maxsum::DomainIterator::getSubInd\(\)](#)
[maxsum::DomainIterator::getInd\(\)](#)

Note: this works by accessing [maxsum::DomainIterator::getSubInd\(\)](#) array. If the domain of the [maxsum::DomainIterator](#) exactly matches the domain of this function, then it is more efficient to index using the corresponding linear index instead. That is

```
maxsum::DiscreteFunction f(...); // some function
maxsum::DomainIterator it(f); // iterator over domain of f
maxsum::ValType x = f(it); // this works
maxsum::ValType x = f(it.getInd()); // more efficient in this case.
```

Parameters

<code>in</code>	<code>it</code>	Iterator used to index this function.
-----------------	-----------------	---------------------------------------

See Also

[maxsum::DomainIterator::getSubInd\(\)](#)
[maxsum::DomainIterator::getInd\(\)](#)

6.3.3.20 `DiscreteFunction & DiscreteFunction::operator= (ValType val)`

Sets this function to a constant scalar value.

Parameters

<code>in</code>	<code>val</code>	the value to assign to this function.
-----------------	------------------	---------------------------------------

6.3.3.21 `DiscreteFunction & DiscreteFunction::operator= (const DiscreteFunction & val)`

Sets this function to be equal to another.

Parameters

<code>in</code>	<code>val</code>	the value to assign to this function.
-----------------	------------------	---------------------------------------

6.3.3.22 **SizeIterator** maxsum::DiscreteFunction::sizeBegin () const [inline]

Returns an iterator to the start of this function's domain variable size list.

This list cannot be modified through this iterator. The values in this list are defined such that $\text{size}[k] = \text{maxsum::getDomainSize}(\text{var}[k])$; However, this method is generally faster than using [maxsum::getDomainSize\(\)](#) directly.

See Also

[maxsum::getDomainSize\(\)](#)
[DiscreteFunction::sizeEnd\(\)](#)

6.3.3.23 **SizeIterator** maxsum::DiscreteFunction::sizeEnd () const [inline]

Returns an iterator to the end of this function's domain variable size list.

This list cannot be modified through this iterator. The values in this list are defined such that $\text{size}[k] = \text{maxsum::getDomainSize}(\text{var}[k])$; However, this method is generally faster than using [maxsum::getDomainSize\(\)](#) directly.

See Also

[maxsum::getDomainSize\(\)](#)
[DiscreteFunction::sizeBegin\(\)](#)

6.3.3.24 void DiscreteFunction::swap (DiscreteFunction & fun)

Swaps the value and domain of this function with another.

After the call to this member function, the elements in this function are those which were in fun before the call, and the elements of fun are those which were in this.

Parameters

<i>fun</i>	Another DiscreteFunction whose value and domain is swapped with that of this one.
------------	---

6.3.3.25 **VarIterator** maxsum::DiscreteFunction::varBegin () const [inline]

Returns an iterator to the start of this function's domain variable list.

The variable list cannot be modified through this iterator.

6.3.3.26 **VarIterator** maxsum::DiscreteFunction::varEnd () const [inline]

Returns an iterator to the end of this function's domain variable list.

This variable list cannot be modified through this iterator.

6.3.4 Friends And Related Function Documentation

6.3.4.1 std::ostream& operator<< (std::ostream & out, const DiscreteFunction & fun) [friend]

Pretty prints this function.

Format is similar to the disp function in Matlab for N-D arrays, except that first dimension appears in rows rather than columns.

The documentation for this class was generated from the following files:

- [include/DiscreteFunction.h](#)
- [src/DiscreteFunction.cpp](#)

6.4 maxsum::DomainConflictException Class Reference

Exception thrown when conflicting domains are specified for a variable.

```
#include <exceptions.h>
```

Public Member Functions

- [DomainConflictException](#) (const std::string where_, const std::string mesg_) throw ()
Creates a new expection of this type.
- const char * [what](#) () const throw ()
Returns a message describing the cause of this exception, and the location it was thrown.
- virtual [~DomainConflictException](#) () throw ()
Destroys this exception and free's its allocated resources.

Protected Attributes

- const std::string [where](#)
String identifying the source code locatioin where this exceptioni was generated.
- const std::string [mesg](#)
Message describing the cause of this exception.

6.4.1 Detailed Description

Exception thrown when conflicting domains are specified for a variable.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 maxsum::DomainConflictException::DomainConflictException (const std::string where_, const std::string mesg_) throw () `[inline]`

Creates a new expection of this type.

Parameters

in	where_	the position in the source code where this was generated.
in	mesg_	message describing the cause of this exception.

6.4.3 Member Function Documentation

6.4.3.1 const char* maxsum::DomainConflictException::what () const throw () `[inline]`

Returns a message describing the cause of this exception, and the location it was thrown.

Returns

a message describing the cause of this exception, and the location it was thrown.

The documentation for this class was generated from the following file:

- include/exceptions.h

6.5 maxsum::DomainIterator Class Reference

This class provides methods for iterating over the Cartesian product for a set of variable domains.

```
#include <DomainIterator.h>
```

Public Types

- typedef std::vector< VarID > VarList
Type of list returned by DomainIterator::getVars()
- typedef std::vector< VallIndex > IndList
Type of list returned by DomainIterator::getSubInd()

Public Member Functions

- void validateRange () const
Utility method that throws an exception if we try to access beyond the end of the domain.
- DomainIterator ()
Default Constructor.
- template<class It >
DomainIterator (It begin, It end)
Construct Domain iterator with initial list of variables.
- DomainIterator (const DiscreteFunction &fun)
Construct Domain Iterator using domain of a given function.
- DomainIterator (const DomainIterator &it)
Copy constructor.
- DomainIterator & operator= (const DomainIterator &it)
Copy assignment.
- bool hasNext () const
Returns true if next call to DomainIterator::operator++() will not throw an exception.
- const IndList & getSubInd () const
Accessor method for current sub indices.
- VallIndex getInd () const
Accessor method for linear index.
- const VarList & getVars () const
Accessor method for variables in domain.
- bool isFixed (VarID var) const
Returns true if this function is conditioned on the specified variable.
- int fixedCount () const
Returns the number of conditioned variables for this function.
- template<class It >
void addVars (It begin, It end)
Add the specified variables to the domain of this iterator.
- DomainIterator & operator++ ()
Increment this iterator to the next element in the domain.
- DomainIterator operator++ (int)
Increment this iterator to the next element in the domain.
- template<class VarIt, class IndIt >
void condition (const VarIt varBegin, const VarIt varEnd, const IndIt indBegin, const IndIt indEnd)

Condition domain on specified variable values.

- void [condition](#) (const [DomainIterator](#) &it)

Convenience function for conditioning one iterator on the current value of another.

6.5.1 Detailed Description

This class provides methods for iterating over the Cartesian product for a set of variable domains.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 maxsum::DomainIterator::DomainIterator () [\[inline\]](#)

Default Constructor.

Creates Iterator over empty domain. Unless [DomainIterator::addVars\(\)](#) is subsequently called [DomainIterator::getVars\(\)](#) and [DomainIterator::getSubInd\(\)](#) will return empty lists, and [DomainIterator::operator++\(\)](#) will throw an exception.

6.5.2.2 template<class It > maxsum::DomainIterator::DomainIterator (It *begin*, It *end*) [\[inline\]](#)

Construct Domain iterator with initial list of variables.

Template Parameters

<i>type</i>	of list iterator used for arguments
-------------	-------------------------------------

Parameters

<i>begin</i>	iterator to first variable in list
<i>end</i>	iterator to end of variable list

6.5.2.3 DomainIterator::DomainIterator (const **DiscreteFunction** & *fun*)

Construct Domain Iterator using domain of a given function.

This is more efficient than creating from scratch.

Parameters

<i>in</i>	<i>fun</i>	Function whose domain should be copied.
-----------	------------	---

6.5.3 Member Function Documentation

6.5.3.1 template<class It > void maxsum::DomainIterator::addVars (It *begin*, It *end*) [\[inline\]](#)

Add the specified variables to the domain of this iterator.

Template Parameters

<i>It</i>	iterator type for input variables.
-----------	------------------------------------

Parameters

<i>begin</i>	iterator to start of variable list to add.
<i>end</i>	iterator to end of variable list to add.

Postcondition

[DomainIterator::getVarS\(\)](#) will return set union of old variable set and input variable set.

[DomainIterator::hasNext\(\)](#) will return true.

Existing conditioned variables will remain conditioned and will have their values preserved. All other variables will have subindex 0.

6.5.3.2 `template<class VarIt , class IndIt > void maxsum::DomainIterator::condition (const VarIt varBegin, const VarIt varEnd, const IndIt indBegin, const IndIt indEnd) [inline]`

Condition domain on specified variable values.

Changes this iterator so that the specified set of variables have fixed values, and are not incremented. The parameters indBegin and indEnd specify the list of values that the corresponding variables should be set to instead. Any variables pointed to by varBegin and varEnd that are not in this iterators domain will be ignored.

Precondition

varBegin and varEnd are iterators to a *sorted* list of [maxsum::VarID](#).

Values in list specified by indBegin and indEnd must be between 0, and their respective variable domain size.

Postcondition

All free indices are set back to 0, and [DomainIterator::hasNext\(\)](#) returns true.

Previously conditioned variables have their values preserved.

Exceptions

OutOfRangeException	is specified condition values are out of range.
-------------------------------------	---

Parameters

in	<i>varBegin</i>	iterator to start of variable list.
in	<i>varEnd</i>	iterator to end of variable list.
in	<i>indBegin</i>	iterator to start of value list.
in	<i>indEnd</i>	iterator to end of value list.

6.5.3.3 `void DomainIterator::condition (const DomainIterator & it)`

Convenience function for conditioning one iterator on the current value of another.

This equivalent to the following code

```
DomainIterator it1(...), it2(...);
DomainIterator::VarList vars = it1.getVars();
DomainIterator::IndList inds = it1.getSubInd();
it2.condition(vars.begin(),vars.end(),inds.begin(), inds.end());
```

See Also

[DomainIterator::condition\(const VarIt,const VarIt,const IndIt,const IndIt\)](#)

6.5.3.4 `ValIndex maxsum::DomainIterator::getInd () const [inline]`

Accessor method for linear index.

Exceptions

<i>maxsum::OutOfRangeException</i>	if iterator is out of range.
--	------------------------------

6.5.3.5 `const IndList& maxsum::DomainIterator::getSubInd () const` `[inline]`

Accessor method for current sub indices.

Exceptions

<i>maxsum::OutOfRangeException</i>	if iterator is out of range.
--	------------------------------

6.5.3.6 `bool maxsum::DomainIterator::hasNext () const` `[inline]`

Returns true if next call to [`DomainIterator::operator++\(\)`](#) will not throw an exception.

This is true provided all free (unconditioned) variables are less than their corresponding maximum: [`maxsum::getDomainSize\(maxsum::VarID\)`](#) - 1.

See Also

[`maxsum::getDomainSize\(maxsum::VarID\)`](#)
[`DomainIterator::condition\(\)`](#)

6.5.3.7 `DomainIterator & DomainIterator::operator++ ()`

Increment this iterator to the next element in the domain.

This is the prefix version: increment first, then return.

Exceptions

<i>OutOfRangeException</i>	if we are already at the last element.
--	--

Returns

iterator to next element

This is the prefix version: increment first, then return.

Exceptions

<i>maxsum::OutOfRangeException</i>	if we are already at the last element.
--	--

Returns

iterator to next element

6.5.3.8 `DomainIterator maxsum::DomainIterator::operator++ (int)` `[inline]`

Increment this iterator to the next element in the domain.

This is the postfix version: return current value, then increment.

Exceptions

<i>OutOfRangeException</i>	if we are already at the last element.
--	--

Returns

iterator to next element

6.5.3.9 void maxsum::DomainIterator::validateRange () const [inline]

Utility method that throws an exception if we try to access beyond the end of the domain.

Exceptions

<i>maxsum::OutOfRangeException</i>	if we've reached the end of the list.
--	---------------------------------------

The documentation for this class was generated from the following files:

- include/[DomainIterator.h](#)
- src/[DomainIterator.cpp](#)

6.6 maxsum::EmptyNoticeException Class Reference

Exception thrown by [maxsum::util::PostOffice::popNotice](#) when there are no active notices.

```
#include <exceptions.h>
```

Public Member Functions

- [EmptyNoticeException](#) (const std::string where_, const std::string mesg_) throw ()
Constructs a new exception with specified location and message.
- const char * [what](#) () const throw ()
Returns a message describing the cause of this exception, and the location it was thrown.
- virtual [~EmptyNoticeException](#) () throw ()
Destroys this exception and free's its allocated resources.

Protected Attributes

- const std::string [where](#)
String identifying the source code locatioin where this exception was generated.
- const std::string [mesg](#)
Message describing the cause of this exception.

6.6.1 Detailed Description

Exception thrown by [maxsum::util::PostOffice::popNotice](#) when there are no active notices.

See Also

[PostOffice::popNotice](#)

6.6.2 Constructor & Destructor Documentation

6.6.2.1 maxsum::EmptyNoticeException::EmptyNoticeException (const std::string *where_*, const std::string *mesg_*) throw ()
[inline]

Constructs a new exception with specified location and message.

Parameters

in	<i>where_</i>	the source code location where this exception was generated.
in	<i>mesg_</i>	Message describing the reason for this exception.

6.6.3 Member Function Documentation

6.6.3.1 const char* maxsum::EmptyNoticeException::what () const throw () [inline]

Returns a message describing the cause of this exception, and the location it was thrown.

Returns

a message describing the cause of this exception, and the location it was thrown.

The documentation for this class was generated from the following file:

- include/exceptions.h

6.7 maxsum::InconsistentDomainException Class Reference

Exception thrown when variable domains are somehow registered as inconsistent.

```
#include <exceptions.h>
```

Public Member Functions

- [InconsistentDomainException](#) (const std::string *where_*, const std::string *mesg_*) throw ()
Creates a new exception of this type.
- const char * [what](#) () const throw ()
Returns a message describing the cause of this exception, and the location it was thrown.
- virtual [~InconsistentDomainException](#) () throw ()
Destroys this exception and free's its allocated resources.

Protected Attributes

- const std::string [where](#)
String identifying the source code location where this exception was generated.
- const std::string [mesg](#)
Message describing the cause of this exception.

6.7.1 Detailed Description

Exception thrown when variable domains are somehow registered as inconsistent.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 `maxsum::InconsistentDomainException::InconsistentDomainException (const std::string where_, const std::string mesg_) throw () [inline]`

Creates a new exception of this type.

Parameters

<code>in</code>	<code>where_</code>	the position in the source code where this was generated.
<code>in</code>	<code>mesg_</code>	message describing the cause of this exception.

6.7.3 Member Function Documentation

6.7.3.1 `const char* maxsum::InconsistentDomainException::what () const throw () [inline]`

Returns a message describing the cause of this exception, and the location it was thrown.

Returns

a message describing the cause of this exception, and the location it was thrown.

The documentation for this class was generated from the following file:

- [include/exceptions.h](#)

6.8 maxsum::util::KeySet< Map > Class Template Reference

Utility class for presenting the keys of a map in a read-only container.

```
#include <util_containers.h>
```

Classes

- class [const_iterator](#)
Iterator type that allows read-only access to the keys of the underlying map.

Public Types

- typedef `Map::key_type` [value_type](#)
The type of value contained in this set, which is always equal to the key type of the underlying map.

Public Member Functions

- [KeySet](#) (const Map *const pKeyMap=0)
Construct a new maxsum::KeySet backed by a specified map.
- void [setMap](#) (const Map *const pKeyMap)
Sets the backend map referenced by this [KeySet](#).
- [const_iterator](#) [begin](#) () const
Returns an iterator to the beginning of the key set.
- [const_iterator](#) [end](#) () const
Returns an iterator to the end of the key set.

- int [size](#) () const
Returns the size of the underlying map.
- [const_iterator find](#) (const [value_type](#) &k) const
Returns an iterator to the specified key, if it is in the map, or KeySet::const_iterator::end if it is not.
- bool [contains](#) (const [value_type](#) &k) const
Returns true if the specified key is in this set.

6.8.1 Detailed Description

```
template<class Map>class maxsum::util::KeySet< Map >
```

Utility class for presenting the keys of a map in a read-only container.

Provides methods for iterating over this set that adhere to standard library concepts.

Attention

This class is used as part of the implementation of the maxsum::PostOffice class only, and so does not need to be referenced directly by calling libraries. We therefore only implement the subset of the standard container interface that we actually need for our own purposes.

Template Parameters

<i>Map</i>	the class of underlying map referenced by this object.
------------	--

6.8.2 Constructor & Destructor Documentation

6.8.2.1 `template<class Map> maxsum::util::KeySet< Map >::KeySet (const Map *const pKeyMap = 0)`
[inline]

Construct a new maxsum::KeySet backed by a specified map.

Parameters

<i>in</i>	<i>pKeyMap</i>	the backend map whoses key set we wish to present through this class. Changes to keyMap are tracked by this object, but no modifications can be made to keyMap through this object.
-----------	----------------	---

Attention

If pKeyMap is null, then it must be set to point to a valid object using [KeySet::setMap](#) before any other member function is called.

Postcondition

*pKeyMap must not be destroyed before this maxsum::KeySet.

6.8.3 Member Function Documentation

6.8.3.1 `template<class Map> const_iterator maxsum::util::KeySet< Map >::begin () const` [inline]

Returns an iterator to the beginning of the key set.

Returns

an iterator to the beginning of the key set.

6.8.3.2 `template<class Map> const_iterator maxsum::util::KeySet< Map >::end () const` `[inline]`

Returns an iterator to the end of the key set.

As in the standard library, the end iterator points one space beyond the last element in the container.

6.8.3.3 `template<class Map> const_iterator maxsum::util::KeySet< Map >::find (const value_type & k) const` `[inline]`

Returns an iterator to the specified key, if it is in the map, or KeySet::const_iterator::end if it is not.

Returns

an iterator to the specified key, if it is in the map, or KeySet::const_iterator::end if it is not.

6.8.3.4 `template<class Map> int maxsum::util::KeySet< Map >::size () const` `[inline]`

Returns the size of the underlying map.

Returns

the size of the underlying map.

The documentation for this class was generated from the following file:

- [include/util_containers.h](#)

6.9 maxsum::MaxSumController Class Reference

This class maintains a factor graph and implements the max-sum algorithm.

```
#include <MaxSumController.h>
```

Public Types

- typedef ValueMap::const_iterator [ConstValueIterator](#)
Read only iterator for all variable value assignments.
- typedef FactorMap::const_iterator [ConstFactorIterator](#)
Read only iterator for all factors.

Public Member Functions

- [MaxSumController](#) (int maxIterations=DEFAULT_MAX_ITERATIONS, ValType maxnorm=DEFAULT_MAX-NORM_THRESHOLD)
Construct a new `maxsum::MaxSumController`.
- void [setFactor](#) (FactorID id, const [DiscreteFunction](#) &factor)
Accessor method for factor function.
- void [removeFactor](#) (FactorID id)
Removes the specified factor from this controller's factor graph.
- void [clear](#) ()
Clear all factors and variables to form an empty factor graph.
- bool [hasFactor](#) (FactorID id) const

- Returns true if and only if the specified factor is managed by this `maxsum::MaxSumController`.*
- `bool hasEdge (FactorID id, VarID var) const`
 - Returns true if and only if the specified variable, `var` and factor, `fac` are known to this `maxsum::MaxSumController`, and `var` is in the domain of `fac`.*
- `int noFactors () const`
 - Returns the number of factors know to this factor graph.*
- `int noVars () const`
 - Returns the number of variables known to this factor graph.*
- `int noEdges () const`
 - Returns the number of edges in the factor graph.*
- `ConstValueIterator valBegin () const`
 - Returns a read only iterator to the beginning of the variable value map.*
- `ConstValueIterator valEnd () const`
 - Returns a read only iterator to the end of the variable value map.*
- `ConstFactorIterator factorBegin () const`
 - Returns a read only iterator to the beginning of the factor map.*
- `ConstFactorIterator factorEnd () const`
 - Returns a read only iterator to the end of the factor map.*
- `const DiscreteFunction & getFactor (FactorID id) const`
 - Accessor method for factor function.*
- `bool hasValue (VarID id) const`
 - Returns true if and only if the specified variable is in the domain of at least one of the factors managed by this `maxsum::MaxSumController`.*
- `ValType getValue (VarID id) const`
 - Returns the current value assigned to the specified variable.*
- `int optimise ()`
 - Runs the max-sum algorithm to optimise the values for each variable.*

Static Public Attributes

- `static const int DEFAULT_MAX_ITERATIONS = 100`
 - Default maximum number of iterations for max-sum algorithm.*
- `static const ValType DEFAULT_MAXNORM_THRESHOLD = 0.0000001`
 - Default maximum maxnorm allowed between the old and new values of a message, before it is assumed to have converged.*

Friends

- `std::ostream & operator<< (std::ostream &out, MaxSumController &controller)`
 - Utility function used to dump the current state of this controller for debugging purposes.*

6.9.1 Detailed Description

This class maintains a factor graph and implements the max-sum algorithm.

More specifically, this class implements the max-sum algorithm using the factor graph regime, in which messages are passed along the edges of a factor graph consisting of factor nodes and variable (action) nodes. Once the algorithm has converged, actions selected by maximising over the sum of messages sent to each variable (action) node.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 `maxsum::MaxSumController::MaxSumController (int maxIterations = DEFAULT_MAX_ITERATIONS, ValType maxnorm = DEFAULT_MAXNORM_THRESHOLD) [inline]`

Construct a new [maxsum::MaxSumController](#).

Parameters

in	<i>maxIterations</i>	The maximum number of iterations for the max-sum algorithm.
in	<i>maxnorm</i>	The maximum maxnorm allowed between the old and new values of a message, before it is assumed to have converged.

6.9.3 Member Function Documentation

6.9.3.1 `void MaxSumController::clear ()`

Clear all factors and variables to form an empty factor graph.

Postcondition

the state of this [maxsum::MaxSumController](#) is reset to that created by the default constructor, with no registered factors or variables.

6.9.3.2 `const DiscreteFunction& maxsum::MaxSumController::getFactor (FactorID id) const [inline]`

Accessor method for factor function.

Parameters

in	<i>id</i>	the unique identifier of the desired factor.
----	-----------	--

Returns

a reference to the function associated with the factor with unique identifier *id*.

Exceptions

maxsum::NoSuchElementException	if the specified factor is not known to this maxsum::MaxSumController .
--	---

6.9.3.3 `ValType maxsum::MaxSumController::getValue (VarID id) const [inline]`

Returns the current value assigned to the specified variable.

This value is only optimised if [MaxSumController::optimise](#) has been previously called, and no factor has been modified since it was last called.

Parameters

in	<i>id</i>	the unique identifier of the desired variable.
----	-----------	--

Returns

the current value assigned to the specified variable.

Exceptions

<i>maxsum::NoSuchElementException</i>	if the specified value is not in the domain of any of the factors managed by this maxsum::MaxSumController .
---	--

6.9.3.4 `bool maxsum::MaxSumController::hasEdge (FactorID id, VarID var) const [inline]`

Returns true if and only if the specified variable, `var` and factor, `fac` are known to this [maxsum::MaxSumController](#), and `var` is in the domain of `fac`.

Returns

true if and only if the specified variable, `var` and factor, `fac` are known to this [maxsum::MaxSumController](#), and `var` is in the domain of `fac`.

6.9.3.5 `bool maxsum::MaxSumController::hasFactor (FactorID id) const [inline]`

Returns true if and only if the specified factor is managed by this [maxsum::MaxSumController](#).

Returns

true if and only if the specified factor is managed by this [maxsum::MaxSumController](#).

6.9.3.6 `bool maxsum::MaxSumController::hasValue (VarID id) const [inline]`

Returns true if and only if the specified variable is in the domain of at least one of the factors managed by this [maxsum::MaxSumController](#).

Returns

true if and only if the specified variable is in the domain of at least one of the factors managed by this [maxsum::MaxSumController](#).

6.9.3.7 `int MaxSumController::optimise ()`

Runs the max-sum algorithm to optimise the values for each variable.

Postcondition

`::getValue(VarID id)` will return the optimal value for the the variable with unique identifier `id`.

Returns

the number of max-sum iterations performed.

Postcondition

[maxsum::MaxSumController::getValue](#) will return the optimal value for the the variable with unique identifier `id`.

6.9.3.8 void MaxSumController::removeFactor (FactorID id)

Removes the specified factor from this controller's factor graph.

In addition, any variables that were previously only connected to this factor, will also be removed from the factor graph. Notice that this does not require the remaining nodes to form a connected graph, but it does mean that variables or factors connect exist in isolation. For example, although every variable and factor must have at least one edge, two variable-factor graph pairs may exist without any edges between the pairs.

Parameters

in	id	the id of the factor to remove from the graph.
----	----	--

Postcondition

If the specified factor was previously in the factor graph, it will be removed by this function.

Any variables that were previously only connected to this factor, will also be removed from the factor graph.

6.9.3.9 void MaxSumController::setFactor (FactorID id, const DiscreteFunction & factor)

Accessor method for factor function.

Parameters

in	id	the unique identifier of the desired factor.
in	factor	the function representing this factor.

Returns

a reference to the function associated with the factor with unique identifier `id`.

Postcondition

A copy of `factor` is stored internally by this [maxsum::MaxSumController](#) and used to form part of a factor graph.

Any previous value of the specified factor is overwritten.

The documentation for this class was generated from the following files:

- [include/MaxSumController.h](#)
- [src/MaxSumController.cpp](#)

6.10 maxsum::NoSuchElementException Class Reference

Exception thrown when there has been an attempt to access an element of a container that does not exist, and cannot be created on demand.

```
#include <exceptions.h>
```

Public Member Functions

- [NoSuchElementException](#) (const std::string where_, const std::string mesg_) throw ()
Constructs a new exception with specified location and message.
- const char * [what](#) () const throw ()
Returns a message describing the cause of this exception, and the location it was thrown.
- virtual [~NoSuchElementException](#) () throw ()
Destroys this exception and free's its allocated resources.

Protected Attributes

- `const std::string where`
String identifying the source code locatioin where this exceptioin was generated.
- `const std::string mesg`
Message describing the cause of this exception.

6.10.1 Detailed Description

Exception thrown when there has been an attempt to access an element of a container that does not exist, and cannot be created on demand.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 `maxsum::NoSuchElementException::NoSuchElementException (const std::string where_, const std::string mesg_) throw () [inline]`

Constructs a new exception with specified location and message.

Parameters

<code>in</code>	<code>where_</code>	the source code location where this exception was generated.
<code>in</code>	<code>mesg_</code>	Message describing the reason for this exception.

6.10.3 Member Function Documentation

6.10.3.1 `const char* maxsum::NoSuchElementException::what () const throw () [inline]`

Returns a message describing the cause of this exception, and the location it was thrown.

Returns

a message describing the cause of this exception, and the location it was thrown.

The documentation for this class was generated from the following file:

- `include/exceptions.h`

6.11 maxsum::OutOfRangeException Class Reference

Exception thrown when indices are out of range.

```
#include <exceptions.h>
```

Public Member Functions

- `OutOfRangeException (const std::string where_, const std::string mesg_) throw ()`
Constructs a new exception with specified location and message.
- `const char * what () const throw ()`
Returns a message describing the cause of this exception, and the location it was thrown.
- `virtual ~OutOfRangeException () throw ()`
Destroys this exception and free's its allocated resources.

Protected Attributes

- `const std::string where`
String identifying the source code locatioin where this exceptioin was generated.
- `const std::string mesg`
Message describing the cause of this exception.

6.11.1 Detailed Description

Exception thrown when indices are out of range.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 `maxsum::OutOfRangeException::OutOfRangeException (const std::string where_, const std::string mesg_) throw ()` `[inline]`

Constructs a new exception with specified location and message.

Parameters

<code>in</code>	<code>where_</code>	the source code location where this exception was generated.
<code>in</code>	<code>mesg_</code>	Message describing the reason for this exception.

6.11.3 Member Function Documentation

6.11.3.1 `const char* maxsum::OutOfRangeException::what () const throw ()` `[inline]`

Returns a message describing the cause of this exception, and the location it was thrown.

Returns

a message describing the cause of this exception, and the location it was thrown.

The documentation for this class was generated from the following file:

- `include/exceptions.h`

6.12 `maxsum::util::PostOffice< Sender, Receiver, Message >` Class Template Reference

Class used to store and manage messages sent between factor graph nodes.

```
#include <PostOffice.h>
```

Public Types

- `typedef util::RefMap< Receiver, Message * > OutMsgMap`
Map of Receivers to Messages for a specific Sender.
- `typedef OutMsgMap::const_iterator OutMsgIt`
Iterator type for `PostOffice::OutMsgMap`.
- `typedef util::RefMap< Sender, Message * > InMsgMap`
Map of Senders to Messages for a specific Receivers.

- typedef [InMsgMap::const_iterator](#) [InMsgIt](#)
Iterator type for [PostOffice::InMsgMap](#).
- typedef [util::KeySet](#)
< [OutboxMap](#) >::const_iterator [SenderIt](#)
Type of iterator returned by [PostOffice::senderBegin\(\)](#) and [PostOffice::senderEnd\(\)](#).
- typedef [util::KeySet](#)< [InboxMap](#) >::const_iterator [ReceiverIt](#)
Type of iterator returned by [PostOffice::receiverBegin](#) and [PostOffice::receiverEnd](#).

Public Member Functions

- [PostOffice](#) ()
Constructs an empty [PostOffice](#) with no edges, senders or receivers.
- void [clear](#) ()
Removes all messages and edges from this postoffice.
- [OutMsgMap](#) [curOutMsgs](#) (Sender s)
Returns the current set of output messages for a given sender.
- [OutMsgMap](#) [prevOutMsgs](#) (Sender s)
Returns the previous set of output messages for a given sender.
- [InMsgMap](#) [curInMsgs](#) (Receiver r)
Returns the current set of input messages for a given receiver.
- [InMsgMap](#) [prevInMsgs](#) (Receiver r)
Returns the previous set of input messages for a given receiver.
- bool [newMail](#) () const
Returns true if any receivers have new mail.
- void [notify](#) (Receiver r)
Notifies a receiver that they have new mail.
- void [notifyAll](#) ()
Notifies all receivers that they have new mail.
- int [noticeCount](#) () const
Returns the number of receivers who are currently notified to check their mail.
- Receiver [popNotice](#) ()
Returns the identity of the next receiver who has new mail, and removes their identity from the notification list.
- void [swapOutBoxes](#) (Sender s)
Swaps the current messages from a specified sender with their previous ones.
- void [addEdge](#) (Sender s, Receiver r)
Adds an edge between a specified sender and receiver.
- void [addEdge](#) (Sender s, Receiver r, const Message &msgVal)
Adds an edge between a specified sender and receiver.
- void [removeEdge](#) (Sender s, Receiver r)
Removes an edge between a specified sender and receiver.
- bool [hasSender](#) (Sender id) const
Returns true if a specified sender is registered with this postoffice.
- bool [hasReceiver](#) (Receiver id) const
Returns true if a specified receiver is registered with this postoffice.
- bool [hasEdge](#) (Sender s, Receiver r) const
Returns true if a specified sender and receiver are connected.
- int [numOfEdges](#) () const
Returns the number of edges (routes between sender's and receivers) registered in this [PostOffice](#).
- int [numOfSenders](#) () const
Returns the number of Sender's registered with this [PostOffice](#).

- `int numOfReceivers () const`
Returns the number of Receiver's registered with this [PostOffice](#).
- `SenderIt senderBegin () const`
Returns iterator to the beginning of the Sender set.
- `SenderIt senderEnd () const`
Returns iterator to the beginning of the Sender set.
- `ReceiverIt receiverBegin ()`
Returns iterator to the beginning of the Receiver set.
- `ReceiverIt receiverEnd ()`
Returns iterator to the beginning of the Receiver set.
- `virtual ~PostOffice ()`
Destructor frees all resources used by this [PostOffice](#).

6.12.1 Detailed Description

```
template<class Sender, class Receiver, class Message = DiscreteFunction>class maxsum::util::PostOffice< Sender, Receiver,
Message >
```

Class used to store and manage messages sent between factor graph nodes.

Template Parameters

<i>Sender</i>	Type used to uniquely identify message senders, e.g. maxsum::FactorID or maxsum::VarID
<i>Receiver</i>	Type used to uniquely identify message receivers, e.g. maxsum::FactorID or maxsum::VarID
<i>Message</i>	Class used to represent messages which, by default, are maxsum::DiscreteFunction objects.

6.12.2 Member Typedef Documentation

6.12.2.1 `template<class Sender , class Receiver , class Message = DiscreteFunction> typedef
util::KeySet<InboxMap>::const_iterator maxsum::util::PostOffice< Sender, Receiver, Message
>::ReceiverIt`

Type of iterator returned by [PostOffice::receiverBegin](#) and [PostOffice::receiverEnd](#).

This type of iterator allows read-only access to the set of Receivers registered with this `maxsum::PostOffice`.

See Also

[PostOffice::receiverBegin](#)
[PostOffice::receiverEnd](#)

6.12.2.2 `template<class Sender , class Receiver , class Message = DiscreteFunction> typedef
util::KeySet<OutboxMap>::const_iterator maxsum::util::PostOffice< Sender, Receiver, Message
>::SenderIt`

Type of iterator returned by [PostOffice::senderBegin\(\)](#) and [PostOffice::senderEnd\(\)](#).

This type of iterator allows read-only access to the set of Senders registered with this `maxsum::PostOffice`.

See Also

[PostOffice::senderBegin](#)
[PostOffice::senderEnd](#)

6.12.3 Constructor & Destructor Documentation

6.12.3.1 `template<class Sender , class Receiver , class Message = DiscreteFunction> virtual maxsum::util::PostOffice< Sender, Receiver, Message >::~~PostOffice () [inline], [virtual]`

Destructor frees all resources used by this [PostOffice](#).

Postcondition

Any external pointers to messages managed by this post office will be invalidated.

6.12.4 Member Function Documentation

6.12.4.1 `template<class Sender , class Receiver , class Message = DiscreteFunction> void maxsum::util::PostOffice< Sender, Receiver, Message >::addEdge (Sender s, Receiver r) [inline]`

Adds an edge between a specified sender and receiver.

Parameters

in	<i>s</i>	the sender's id
in	<i>r</i>	the receiver's id

Postcondition

Messages can now be sent from *s* to *s* via this maxsum::PostOffice.

6.12.4.2 `template<class Sender , class Receiver , class Message = DiscreteFunction> void maxsum::util::PostOffice< Sender, Receiver, Message >::addEdge (Sender s, Receiver r, const Message & msgVal) [inline]`

Adds an edge between a specified sender and receiver.

Parameters

in	<i>s</i>	the sender's id
in	<i>r</i>	the receiver's id
in	<i>msgVal</i>	initial value for newly constructed messages.

Postcondition

Messages can now be sent from *s* to *s* via this maxsum::PostOffice.

6.12.4.3 `template<class Sender , class Receiver , class Message = DiscreteFunction> InMsgMap maxsum::util::PostOffice< Sender, Receiver, Message >::curlnMsgs (Receiver r) [inline]`

Returns the current set of input messages for a given receiver.

Exceptions

maxsum::UnknownAddress-Exception	if receiver is not registered.
--	--------------------------------

6.12.4.4 `template<class Sender , class Receiver , class Message = DiscreteFunction> OutMsgMap
maxsum::util::PostOffice< Sender, Receiver, Message >::curOutMsgs (Sender s) [inline]`

Returns the current set of output messages for a given sender.

Exceptions

<i>maxsum::UnknownAddress-Exception</i>	if sender is not registered.
---	------------------------------

6.12.4.5 `template<class Sender , class Receiver , class Message = DiscreteFunction> bool maxsum::util::PostOffice<
Sender, Receiver, Message >::hasEdge (Sender s, Receiver r) const [inline]`

Returns true if a specified sender and receiver are connected.

Parameters

in	<i>s</i>	the sender's id
in	<i>r</i>	the receiver's id

Returns

true if *s* can send mail to *r* via this maxsum::PostOffice.

6.12.4.6 `template<class Sender , class Receiver , class Message = DiscreteFunction> void maxsum::util::PostOffice<
Sender, Receiver, Message >::notify (Receiver r) [inline]`

Notifies a receiver that they have new mail.

Parameters

in	<i>r</i>	the receiver to notify
----	----------	------------------------

6.12.4.7 `template<class Sender , class Receiver , class Message = DiscreteFunction> Receiver
maxsum::util::PostOffice< Sender, Receiver, Message >::popNotice () [inline]`

Returns the identity of the next receiver who has new mail, and removes their identity from the notification list.

Precondition

PostOffice::newmail returns true
[*PostOffice::noticeCount*](#) returns integer greater than 0.

Postcondition

the size of the notification list is reduced by 1.

Exceptions

<i>maxsum::EmptyNotice-Exception</i>	if <i>PostOffice::newMail</i> returns false;
--	--

6.12.4.8 `template<class Sender , class Receiver , class Message = DiscreteFunction> InMsgMap
maxsum::util::PostOffice< Sender, Receiver, Message >::prevInMsgs (Receiver r) [inline]`

Returns the previous set of input messages for a given receiver.

Exceptions

<i>maxsum::UnknownAddress-Exception</i>	if receiver is not registered.
---	--------------------------------

6.12.4.9 `template<class Sender , class Receiver , class Message = DiscreteFunction> OutMsgMap
maxsum::util::PostOffice< Sender, Receiver, Message >::prevOutMsgs (Sender s) [inline]`

Returns the previous set of output messages for a given sender.

Exceptions

<i>maxsum::UnknownAddress-Exception</i>	if sender is not registered.
---	------------------------------

6.12.4.10 `template<class Sender , class Receiver , class Message = DiscreteFunction> ReceiverIt
maxsum::util::PostOffice< Sender, Receiver, Message >::receiverBegin () [inline]`

Returns iterator to the beginning of the Receiver set.

Together with [PostOffice::receiverEnd](#) this provides access to the set of all receivers. Example usage:

```
PostOffice<VarID,FactorID> post (...);
...
typedef PostOffice<VarID,FactorID>::ReceiverIt Iterator;
for(Iterator it=post.receiverBegin(); it!=post.receiverEnd(); ++it)
{
    std::cout << "Receiver " << *it << " is registered.\n";
}
```

See Also

[PostOffice::receiverEnd](#)

6.12.4.11 `template<class Sender , class Receiver , class Message = DiscreteFunction> ReceiverIt
maxsum::util::PostOffice< Sender, Receiver, Message >::receiverEnd () [inline]`

Returns iterator to the beginning of the Receiver set.

Together with [PostOffice::receiverBegin](#) this provides access to the set of all receivers. Example usage:

```
PostOffice<VarID,FactorID> post (...);
...
typedef PostOffice<VarID,FactorID>::ReceiverIt Iterator;
for(Iterator it=post.receiverBegin(); it!=post.receiverEnd(); ++it)
{
    std::cout << "Receiver " << *it << " is registered.\n";
}
```

See Also

[PostOffice::receiverBegin](#)

6.12.4.12 `template<class Sender , class Receiver , class Message = DiscreteFunction> void maxsum::util::PostOffice< Sender, Receiver, Message >::removeEdge (Sender s, Receiver r) [inline]`

Removes an edge between a specified sender and receiver.

Parameters

in	<i>s</i>	the sender's id
in	<i>r</i>	the receiver's id

Postcondition

Messages will no longer be sent from *s* to *s* via this maxsum::PostOffice.

6.12.4.13 `template<class Sender , class Receiver , class Message = DiscreteFunction> SenderIt maxsum::util::PostOffice< Sender, Receiver, Message >::senderBegin () const [inline]`

Returns iterator to the beginning of the Sender set.

Together with [PostOffice::senderEnd](#) this provides access to the set of all senders. Example usage:

```
PostOffice<VarID,FactorID> post (...);
...
typedef PostOffice<VarID,FactorID>::SenderIt Iterator;
for(Iterator it=post.senderBegin(); it!=post.senderEnd(); ++it)
{
    std::cout << "Sender " << *it << " is registered.\n";
}
```

See Also

[PostOffice::senderEnd](#)

6.12.4.14 `template<class Sender , class Receiver , class Message = DiscreteFunction> SenderIt maxsum::util::PostOffice< Sender, Receiver, Message >::senderEnd () const [inline]`

Returns iterator to the beginning of the Sender set.

Together with [PostOffice::senderBegin](#) this provides access to the set of all senders. Example usage:

```
PostOffice<VarID,FactorID> post (...);
...
typedef PostOffice<VarID,FactorID>::SenderIt Iterator;
for(Iterator it=post.senderBegin(); it!=post.senderEnd(); ++it)
{
    std::cout << "Sender " << *it << " is registered.\n";
}
```

See Also

[PostOffice::senderBegin](#)

6.12.4.15 `template<class Sender , class Receiver , class Message = DiscreteFunction> void maxsum::util::PostOffice< Sender, Receiver, Message >::swapOutBoxes (Sender s) [inline]`

Swaps the current messages from a specified sender with their previous ones.

This allows the previous messages to be overwritten with new messages, without having to free or allocate temporary objects.

The documentation for this class was generated from the following file:

- include/PostOffice.h

6.13 maxsum::util::RefMap< Key, Val > Class Template Reference

This class provides a read only wrapper around an existing map, such as that provided by std::map.

```
#include <util_containers.h>
```

Public Types

- typedef std::map< Key, Val > [PtrMap](#)
Type of map used to store value pointers.
- typedef PtrMap::const_iterator [const_iterator](#)
Iterator type returned by this map.
- typedef PtrMap::value_type [value_type](#)
Key-Value pair type stored by this map.
- typedef PtrMap::value_type [key_type](#)
Key type used to index this map.
- typedef PtrMap::mapped_type [mapped_type](#)
Value type stored by this map.

Public Member Functions

- [RefMap](#) (const [PtrMap](#) &map)
Constructs an empty map.
- [const_iterator begin](#) () const
Returns an iterator the first element in this map.
- [const_iterator end](#) () const
Returns an iterator to the next position after the last element of this map.
- const Val & [operator\[\]](#) (Key key) const
Provides a constant reference to the value pointed to by key, or throws an exception is the key is not currently mapped to any value.
- [const_iterator find](#) (Key key) const
Tries to find the value for a given key in this map.
- int [size](#) () const
Returns the number of elements in this map.

6.13.1 Detailed Description

```
template<class Key, class Val>class maxsum::util::RefMap< Key, Val >
```

This class provides a read only wrapper around an existing map, such as that provided by std::map.

In particular, this class provides a const implementation of the [] operator, which throws an exception if the requested key is not already in the underlying map, rather than trying to insert it. This functionality is not provided by the standard library.

Note: Currently, we only implement that subset of the map interface that we actually require for our purposes.

Attention

This class is used as part of the implementation of the `maxsum::PostOffice` class only, and so does not need to be referenced directly by calling libraries. We therefore only implement the subset of the standard container interface that we actually need for our own purposes.

Template Parameters

<i>Key</i>	Type of key used by this map.
<i>Val</i>	Type of value referenced by this map. Internally, the map stores pointers to <code>Val</code> objects, rather than the <code>Val</code> objects themselves, but presents the same interface as a <code>std::map</code> that does store them.

6.13.2 Member Function Documentation

6.13.2.1 `template<class Key , class Val > const_iterator maxsum::util::RefMap< Key, Val >::begin () const`
`[inline]`

Returns an iterator the first element in this map.

This iterator provides readonly access to the elements of this map.

6.13.2.2 `template<class Key , class Val > const_iterator maxsum::util::RefMap< Key, Val >::end () const`
`[inline]`

Returns an iterator to the next position after the last element of this map.

See C++ standard library documentation for rationale.

6.13.2.3 `template<class Key , class Val > const Val& maxsum::util::RefMap< Key, Val >::operator[] (Key key) const`
`[inline]`

Provides a constant reference to the value pointed to by `key`, or throws an exception is the `key` is not currently mapped to any value.

Attention

This is intentionally different from the behaviour of `std::map::operator[]`, because we only want to provide read-access through this operator.

Parameters

<i>in</i>	<i>key</i>	the key whoses value we want to return.
-----------	------------	---

Exceptions

<i>NoSuchElementException</i>	if key is not currently in this map.
---	--------------------------------------

Returns

a constant reference to the mapped value.

The documentation for this class was generated from the following file:

- [include/util_containers.h](#)

6.14 maxsum::UnknownAddressException Class Reference

Exception thrown when a [maxsum::util::PostOffice](#) does not recognise the ID of a Sender or Receiver.

```
#include <exceptions.h>
```

Public Member Functions

- [UnknownAddressException](#) (const std::string where_, const std::string mesg_) throw ()
Constructs a new exception with specified location and message.
- const char * [what](#) () const throw ()
Returns a message describing the cause of this exception, and the location it was thrown.
- virtual ~[UnknownAddressException](#) () throw ()
Destroys this exception and free's its allocated resources.

Protected Attributes

- const std::string [where](#)
String identifying the source code locatioin where this exception was generated.
- const std::string [mesg](#)
Message describing the cause of this exception.

6.14.1 Detailed Description

Exception thrown when a [maxsum::util::PostOffice](#) does not recognise the ID of a Sender or Receiver.

6.14.2 Constructor & Destructor Documentation

6.14.2.1 `maxsum::UnknownAddressException::UnknownAddressException (const std::string where_, const std::string mesg_) throw () [inline]`

Constructs a new exception with specified location and message.

Parameters

in	<i>where_</i>	the source code location where this exception was generated.
in	<i>mesg_</i>	Message describing the reason for this exception.

6.14.3 Member Function Documentation

6.14.3.1 `const char* maxsum::UnknownAddressException::what () const throw () [inline]`

Returns a message describing the cause of this exception, and the location it was thrown.

Returns

a message describing the cause of this exception, and the location it was thrown.

The documentation for this class was generated from the following file:

- include/[exceptions.h](#)

6.15 maxsum::UnknownVariableException Class Reference

Exception thrown when a variable is referred to, but has not yet been registered using either [maxsum::registerVariable](#) or [maxsum::registerVariables](#).

```
#include <exceptions.h>
```

Public Member Functions

- [UnknownVariableException](#) (const std::string where_, const std::string msg_) throw ()
Creates a new exception of this type.
- const char * [what](#) () const throw ()
Returns a message describing the cause of this exception, and the location it was thrown.
- virtual [~UnknownVariableException](#) () throw ()
Destroys this exception and free's its allocated resources.

Protected Attributes

- const std::string [where](#)
String identifying the source code location where this exception was generated.
- const std::string [msg](#)
Message describing the cause of this exception.

6.15.1 Detailed Description

Exception thrown when a variable is referred to, but has not yet been registered using either [maxsum::registerVariable](#) or [maxsum::registerVariables](#).

6.15.2 Constructor & Destructor Documentation

6.15.2.1 `maxsum::UnknownVariableException::UnknownVariableException (const std::string where_, const std::string msg_) throw () [inline]`

Creates a new exception of this type.

Parameters

in	<i>where_</i>	the position in the source code where this was generated.
in	<i>msg_</i>	message describing the cause of this exception.

6.15.3 Member Function Documentation

6.15.3.1 `const char* maxsum::UnknownVariableException::what () const throw () [inline]`

Returns a message describing the cause of this exception, and the location it was thrown.

Returns

a message describing the cause of this exception, and the location it was thrown.

The documentation for this class was generated from the following file:

- include/[exceptions.h](#)

Chapter 7

File Documentation

7.1 include/common.h File Reference

Common types and functions used by Max-Sum library.

```
#include <cfloat>
#include <string>
#include <vector>
#include "exceptions.h"
```

Namespaces

- namespace [maxsum](#)
Namespace for all public types and functions defined by the Max-Sum library.

Typedefs

- typedef double [maxsum::ValType](#)
Type of values stored by [maxsum::DiscreteFunction](#) objects.
- typedef unsigned int [maxsum::VarID](#)
Type used for uniquely identifying variables.
- typedef unsigned int [maxsum::FactorID](#)
Type used for uniquely identifying factors in a factor graph.
- typedef int [maxsum::ValIndex](#)
Integer type used for indexing coefficient values.

Functions

- template<class VecType >
void [maxsum::ind2sub](#) (const VecType &siz, const typename VecType::value_type ind, VecType &sub)
C++ Implementation of Matlab ind2sub function.
- template<class SizIt, class SubIt >
ValIndex [maxsum::sub2ind](#) (SizIt sizFirst, SizIt sizEnd, SubIt subFirst, SubIt subEnd)
C++ Implementation of Matlab sub2ind function.
- template<class VecType >
VecType::value_type [maxsum::sub2ind](#) (const VecType &siz, const VecType &sub)
C++ Implementation of Matlab sub2ind function.

Variables

- const ValType `maxsum::DEFAULT_VALUE_TOLERANCE` = `DBL_EPSILON * 1000.0`
Default tolerance used for comparing values of type `maxsum::ValType`.

7.1.1 Detailed Description

Common types and functions used by Max-Sum library. In particular, this file defines common typedefs, and functions for calculating subindices and linear indices for referencing the contents of N-D arrays. These resemble similar functions defined in the Matlab environment.

Author

Luke Teacy

7.2 include/DiscreteFunction.h File Reference

Defines the `maxsum::DiscreteFunction` class and related utility functions.

```
#include <iostream>
#include <cassert>
#include <algorithm>
#include "common.h"
#include "register.h"
#include "DomainIterator.h"
```

Classes

- class `maxsum::DiscreteFunction`
Class representing functions of sets of variables with discrete domains.

Namespaces

- namespace `maxsum`
Namespace for all public types and functions defined by the Max-Sum library.

Functions

- `std::ostream & maxsum::operator<<` (`std::ostream &out`, const `DiscreteFunction` &fun)
Pretty prints a `maxsum::DiscreteFunction` Format is similar to the `disp` function in Matlab for N-D arrays, except that first dimension appears in rows rather than columns.
- `bool maxsum::sameDomain` (const `DiscreteFunction` &f1, const `DiscreteFunction` &f2)
Check that two `maxsum::DiscreteFunction` objects have the same domain.
- `bool maxsum::equalWithinTolerance` (const `DiscreteFunction` &f1, const `DiscreteFunction` &f2, `ValType` tol=`DEFAULT_VALUE_TOLERANCE`)
Check that two `maxsum::DiscreteFunction` objects are equal within a specified tolerance.
- `bool maxsum::strictlyEqualWithinTolerance` (const `DiscreteFunction` &f1, const `DiscreteFunction` &f2, `ValType` tol=`DEFAULT_VALUE_TOLERANCE`)
Check that two `maxsum::DiscreteFunction` objects are equal with a specified tolerance, and have exactly the same domain.
- `bool maxsum::operator==` (const `DiscreteFunction` &f1, const `DiscreteFunction` &f2)

- Return true if functions are equal.*
- bool `maxsum::operator!=` (const DiscreteFunction &f1, const DiscreteFunction &f2)
Return true if functions are equal.
- template<typename F >
void `maxsum::marginal` (const DiscreteFunction &inFun, F aggregate, DiscreteFunction &outFun)
Marginalise a `maxsum::DiscreteFunction` using a specified aggregation function.
- void `maxsum::maxMarginal` (const DiscreteFunction &inFun, DiscreteFunction &outFun)
Marginalise a `maxsum::DiscreteFunction` by maximisation.
- void `maxsum::minMarginal` (const DiscreteFunction &inFun, DiscreteFunction &outFun)
Marginalise a `maxsum::DiscreteFunction` by minimisation.
- void `maxsum::meanMarginal` (const DiscreteFunction &inFun, DiscreteFunction &outFun)
Marginalise a `maxsum::DiscreteFunction` by averaging.

7.2.1 Detailed Description

Defines the `maxsum::DiscreteFunction` class and related utility functions.

7.3 include/DomainIterator.h File Reference

This file defines the `maxsum::DomainIterator` class.

```
#include <algorithm>
#include <vector>
#include "common.h"
#include "register.h"
```

Classes

- class `maxsum::DomainIterator`
This class provides methods for iterating over the Cartesian product for a set of variable domains.

Namespaces

- namespace `maxsum`
Namespace for all public types and functions defined by the Max-Sum library.

7.3.1 Detailed Description

This file defines the `maxsum::DomainIterator` class. The `maxsum::DomainIterator` class provides methods for iterating over the Cartesian product for a set of variable domains.

7.4 include/exceptions.h File Reference

Defines exception classes for all error conditions that may occur in the maxsum library.

```
#include <string>
#include <exception>
```

Classes

- class [maxsum::NoSuchElementException](#)
Exception thrown when there has been an attempt to access an element of a container that does not exist, and cannot be created on demand.
- class [maxsum::EmptyNoticeException](#)
Exception thrown by [maxsum::util::PostOffice::popNotice](#) when there are no active notices.
- class [maxsum::UnknownAddressException](#)
Exception thrown when a [maxsum::util::PostOffice](#) does not recognise the ID of a Sender or Receiver.
- class [maxsum::BadDomainException](#)
Exception thrown when subindices are incorrectly specified for a [maxsum::DiscreteFunction](#).
- class [maxsum::OutOfRangeException](#)
Exception thrown when indices are out of range.
- class [maxsum::DomainConflictException](#)
Exception thrown when conflicting domains are specified for a variable.
- class [maxsum::UnknownVariableException](#)
Exception thrown when a variable is referred to, but has not yet been registered using either [maxsum::registerVariable](#) or [maxsum::registerVariables](#).
- class [maxsum::InconsistentDomainException](#)
Exception thrown when variable domains are somehow registered as inconsistent.

Namespaces

- namespace [maxsum](#)
Namespace for all public types and functions defined by the Max-Sum library.

7.4.1 Detailed Description

Defines exception classes for all error conditions that may occur in the maxsum library.

Author

Luke Teacy

7.5 include/mainpage.h File Reference

This header defines the main page for this documentation, and is for that purpose only.

Namespaces

- namespace [maxsum](#)
Namespace for all public types and functions defined by the Max-Sum library.

7.5.1 Detailed Description

This header defines the main page for this documentation, and is for that purpose only.

Author

Dr W. T. Luke Teacy, University of Southampton

7.6 include/MaxSumController.h File Reference

Defines the MaxSumController class, which implements the max-sum algorithm.

```
#include <iostream>
#include <map>
#include <stack>
#include "common.h"
#include "DiscreteFunction.h"
#include "PostOffice.h"
```

Classes

- class [maxsum::MaxSumController](#)

This class maintains a factor graph and implements the max-sum algorithm.

Namespaces

- namespace [maxsum](#)

Namespace for all public types and functions defined by the Max-Sum library.

Functions

- std::ostream & [maxsum::operator<<](#) (std::ostream &out, [MaxSumController](#) &controller)

Utility function used to dump the current state of this controller for debugging purposes.

7.6.1 Detailed Description

Defines the MaxSumController class, which implements the max-sum algorithm.

Author

W. T. Luke Teacy - University of Southampton

7.7 include/PostOffice.h File Reference

This file defines the utility maxsum::PostOffice class.

```
#include <map>
#include <set>
#include <queue>
#include "DiscreteFunction.h"
#include "util_containers.h"
```

Classes

- class [maxsum::util::PostOffice< Sender, Receiver, Message >](#)

Class used to store and manage messages sent between factor graph nodes.

Namespaces

- namespace [maxsum](#)
Namespace for all public types and functions defined by the Max-Sum library.
- namespace [maxsum::util](#)
Utility namespace for types used for maxsum library implementation.

Typedefs

- typedef PostOffice< VarID,
FactorID > [maxsum::util::V2FPostOffice](#)
Convenience typedef for Variable to Factor PostOffices.
- typedef PostOffice< FactorID,
VarID > [maxsum::util::F2VPostOffice](#)
Convenience typedef for Factor to Variable PostOffices.
- typedef PostOffice< FactorID,
FactorID > [maxsum::util::F2FPostOffice](#)
Convenience typedef for Factor to Factor PostOffices.

7.7.1 Detailed Description

This file defines the utility `maxsum::PostOffice` class.

7.8 include/register.h File Reference

This Header defines functions for registering the set of all variables on which [maxsum::DiscreteFunction](#) objects may depend.

```
#include "common.h"
```

Namespaces

- namespace [maxsum](#)
Namespace for all public types and functions defined by the Max-Sum library.

Functions

- bool [maxsum::isRegistered](#) (VarID var)
Returns true if the specified variable is registered.
- template<class VarIt >
bool [maxsum::allRegistered](#) (VarIt varBegin, VarIt varEnd)
Returns true if all specified variables are registered.
- ValIndex [maxsum::getDomainSize](#) (VarID var)
Returns the registered domain size for a specified variable.
- int [maxsum::getNumOfRegisteredVariables](#) ()
Returns the number of currently registered variables.
- void [maxsum::registerVariable](#) (VarID var, ValIndex siz)
Registers a variable with a specified domain size.
- template<class VarIt, class IndIt >
void [maxsum::registerVariables](#) (VarIt varBegin, VarIt varEnd, IndIt sizBegin, IndIt sizEnd)
Register a list of variables with specified domain sizes.

7.8.1 Detailed Description

This Header defines functions for registering the set of all variables on which `maxsum::DiscreteFunction` objects may depend. Variables are uniquely identified by a key of type `maxsum::VarID` and each as a specified domain size. The purpose of the functions defined in this file are to register the domain size for each variable before it is used, and ensure that this size remains fixed throughout a programs execution. Variables can be registered multiple times, but in each case the domain size must not change. Variables must always be registered before they are referenced by any function.

7.9 include/util_containers.h File Reference

This file defines some utility container types required for the implementation of the maxsum library.

```
#include <cassert>
#include <map>
```

Classes

- class `maxsum::util::KeySet< Map >`
Utility class for presenting the keys of a map in a read-only container.
- class `maxsum::util::KeySet< Map >::const_iterator`
Iterator type that allows read-only access to the keys of the underlying map.
- class `maxsum::util::RefMap< Key, Val >`
This class provides a read only wrapper around an existing map, such as that provided by `std::map`.

Namespaces

- namespace `maxsum`
Namespace for all public types and functions defined by the Max-Sum library.
- namespace `maxsum::util`
Utility namespace for types used for maxsum library implementation.

7.9.1 Detailed Description

This file defines some utility container types required for the implementation of the maxsum library.

Attention

The types defined in this header are not intended to form part of the public interface to the maxsum library. Knowledge of these types is therefore not required for interfacing with the library.

7.10 src/DiscreteFunction.cpp File Reference

Implementation of member functions of class `maxsum::DiscreteFunction`.

```
#include <cmath>
#include <vector>
#include <cstdarg>
#include <algorithm>
#include <iostream>
#include "DiscreteFunction.h"
#include "DomainIterator.h"
```

7.10.1 Detailed Description

Implementation of member functions of class [maxsum::DiscreteFunction](#).

7.11 src/DomainIterator.cpp File Reference

This file implements the [maxsum::DomainIterator](#) class.

```
#include "register.h"
#include "DomainIterator.h"
#include "DiscreteFunction.h"
```

7.11.1 Detailed Description

This file implements the [maxsum::DomainIterator](#) class. The [maxsum::DomainIterator](#) class provides methods for iterating over the Cartesian product for a set of variable domains.

7.12 src/MaxSumController.cpp File Reference

Implementation of MaxSumController class members.

```
#include "MaxSumController.h"
#include <iostream>
```

7.12.1 Detailed Description

Implementation of MaxSumController class members.

See Also

[MaxSumController.h](#)

Author

W. T. Luke Teacy - University of Southampton

7.13 src/register.cpp File Reference

Implementation of functions in [register.h](#).

```
#include <map>
#include <string>
#include <sstream>
#include "register.h"
```

7.13.1 Detailed Description

Implementation of functions in [register.h](#).

See Also

[register.h](#)

Author

Luke Teacy

Index

- ~PostOffice
 - maxsum::util::PostOffice, [59](#)
- add
 - maxsum::DiscreteFunction, [34](#)
- addEdge
 - maxsum::util::PostOffice, [59](#)
- addVars
 - maxsum::DomainIterator, [43](#)
- allRegistered
 - maxsum, [20](#)
- at
 - maxsum::DiscreteFunction, [34](#), [35](#)
- BadDomainException
 - maxsum::BadDomainException, [29](#)
- begin
 - maxsum::util::KeySet, [49](#)
 - maxsum::util::RefMap, [64](#)
- clear
 - maxsum::MaxSumController, [52](#)
- condition
 - maxsum::DiscreteFunction, [35](#)
 - maxsum::DomainIterator, [44](#)
- curlInMsgs
 - maxsum::util::PostOffice, [59](#)
- curOutMsgs
 - maxsum::util::PostOffice, [59](#)
- dependsOn
 - maxsum::DiscreteFunction, [36](#)
- DiscreteFunction
 - maxsum::DiscreteFunction, [33](#), [34](#)
- DomainConflictException
 - maxsum::DomainConflictException, [41](#)
- DomainIterator
 - maxsum::DomainIterator, [43](#)
- EmptyNoticeException
 - maxsum::EmptyNoticeException, [47](#)
- end
 - maxsum::util::KeySet, [49](#)
 - maxsum::util::RefMap, [64](#)
- equalWithinTolerance
 - maxsum, [20](#)
- expand
 - maxsum::DiscreteFunction, [36](#), [37](#)
- FactorID
 - maxsum, [19](#)
- find
 - maxsum::util::KeySet, [50](#)
- getDomainSize
 - maxsum, [20](#)
- getFactor
 - maxsum::MaxSumController, [52](#)
- getInd
 - maxsum::DomainIterator, [44](#)
- getNumOfRegisteredVariables
 - maxsum, [21](#)
- getSubInd
 - maxsum::DomainIterator, [45](#)
- getValue
 - maxsum::MaxSumController, [52](#)
- hasEdge
 - maxsum::MaxSumController, [53](#)
 - maxsum::util::PostOffice, [60](#)
- hasFactor
 - maxsum::MaxSumController, [53](#)
- hasNext
 - maxsum::DomainIterator, [45](#)
- hasValue
 - maxsum::MaxSumController, [53](#)
- include/DiscreteFunction.h, [68](#)
- include/DomainIterator.h, [69](#)
- include/MaxSumController.h, [71](#)
- include/PostOffice.h, [71](#)
- include/common.h, [67](#)
- include/exceptions.h, [69](#)
- include/mainpage.h, [70](#)
- include/register.h, [72](#)
- include/util_containers.h, [73](#)
- InconsistentDomainException
 - maxsum::InconsistentDomainException, [48](#)
- ind2sub
 - maxsum, [21](#)
- isRegistered
 - maxsum, [21](#)
- KeySet
 - maxsum::util::KeySet, [49](#)
- marginal
 - maxsum, [22](#)
- maxMarginal
 - maxsum, [23](#)
- MaxSumController
 - maxsum::MaxSumController, [52](#)

- maxnorm
 - maxsum::DiscreteFunction, 37
- maxsum, 17
 - allRegistered, 20
 - equalWithinTolerance, 20
 - FactorID, 19
 - getDomainSize, 20
 - getNumOfRegisteredVariables, 21
 - ind2sub, 21
 - isRegistered, 21
 - marginal, 22
 - maxMarginal, 23
 - meanMarginal, 23
 - minMarginal, 24
 - operator<<, 25
 - operator==, 25
 - registerVariable, 25
 - registerVariables, 26
 - sameDomain, 26
 - strictlyEqualWithinTolerance, 27
 - sub2ind, 27
 - VallIndex, 19
 - ValType, 19
 - VarID, 19
- maxsum::BadDomainException, 29
 - BadDomainException, 29
 - what, 30
- maxsum::DiscreteFunction, 30
 - add, 34
 - at, 34, 35
 - condition, 35
 - dependsOn, 36
 - DiscreteFunction, 33, 34
 - expand, 36, 37
 - maxnorm, 37
 - min, 37
 - operator<<, 40
 - operator(), 37–39
 - operator=, 39
 - sizeBegin, 39
 - sizeEnd, 40
 - swap, 40
 - varBegin, 40
 - varEnd, 40
- maxsum::DomainConflictException, 41
 - DomainConflictException, 41
 - what, 41
- maxsum::DomainIterator, 42
 - addVars, 43
 - condition, 44
 - DomainIterator, 43
 - getInd, 44
 - getSubInd, 45
 - hasNext, 45
 - operator++, 45
 - validateRange, 46
- maxsum::EmptyNoticeException, 46
 - EmptyNoticeException, 47
- what, 47
- maxsum::InconsistentDomainException, 47
 - InconsistentDomainException, 48
 - what, 48
- maxsum::MaxSumController, 50
 - clear, 52
 - getFactor, 52
 - getValue, 52
 - hasEdge, 53
 - hasFactor, 53
 - hasValue, 53
 - MaxSumController, 52
 - optimise, 53
 - removeFactor, 53
 - setFactor, 54
- maxsum::NoSuchElementException, 54
 - NoSuchElementException, 55
 - what, 55
- maxsum::OutOfRangeException, 55
 - OutOfRangeException, 56
 - what, 56
- maxsum::UnknownAddressException, 65
 - UnknownAddressException, 65
 - what, 65
- maxsum::UnknownVariableException, 66
 - UnknownVariableException, 66
 - what, 66
- maxsum::util, 28
- maxsum::util::KeySet
 - begin, 49
 - end, 49
 - find, 50
 - KeySet, 49
 - size, 50
- maxsum::util::KeySet< Map >, 48
- maxsum::util::KeySet< Map >::const_iterator, 30
- maxsum::util::PostOffice
 - ~PostOffice, 59
 - addEdge, 59
 - curlnMsgs, 59
 - curOutMsgs, 59
 - hasEdge, 60
 - notify, 60
 - popNotice, 60
 - prevInMsgs, 60
 - prevOutMsgs, 61
 - receiverBegin, 61
 - receiverEnd, 61
 - ReceiverIt, 58
 - removeEdge, 61
 - senderBegin, 62
 - senderEnd, 62
 - SenderIt, 58
 - swapOutBoxes, 62
- maxsum::util::PostOffice< Sender, Receiver, Message >, 56
- maxsum::util::RefMap
 - begin, 64

- end, 64
- maxsum::util::RefMap< Key, Val >, 63
- meanMarginal
 - maxsum, 23
- min
 - maxsum::DiscreteFunction, 37
- minMarginal
 - maxsum, 24
- NoSuchElementException
 - maxsum::NoSuchElementException, 55
- notify
 - maxsum::util::PostOffice, 60
- operator<<
 - maxsum, 25
 - maxsum::DiscreteFunction, 40
- operator()
 - maxsum::DiscreteFunction, 37–39
- operator++
 - maxsum::DomainIterator, 45
- operator=
 - maxsum::DiscreteFunction, 39
- operator==
 - maxsum, 25
- optimise
 - maxsum::MaxSumController, 53
- OutOfRangeException
 - maxsum::OutOfRangeException, 56
- popNotice
 - maxsum::util::PostOffice, 60
- prevInMsgs
 - maxsum::util::PostOffice, 60
- prevOutMsgs
 - maxsum::util::PostOffice, 61
- receiverBegin
 - maxsum::util::PostOffice, 61
- receiverEnd
 - maxsum::util::PostOffice, 61
- ReceiverIt
 - maxsum::util::PostOffice, 58
- registerVariable
 - maxsum, 25
- registerVariables
 - maxsum, 26
- removeEdge
 - maxsum::util::PostOffice, 61
- removeFactor
 - maxsum::MaxSumController, 53
- sameDomain
 - maxsum, 26
- senderBegin
 - maxsum::util::PostOffice, 62
- senderEnd
 - maxsum::util::PostOffice, 62
- SenderIt
 - maxsum::util::PostOffice, 58
- setFactor
 - maxsum::MaxSumController, 54
- size
 - maxsum::util::KeySet, 50
- sizeBegin
 - maxsum::DiscreteFunction, 39
- sizeEnd
 - maxsum::DiscreteFunction, 40
- src/DiscreteFunction.cpp, 73
- src/DomainIterator.cpp, 74
- src/MaxSumController.cpp, 74
- src/register.cpp, 74
- strictlyEqualWithinTolerance
 - maxsum, 27
- sub2ind
 - maxsum, 27
- swap
 - maxsum::DiscreteFunction, 40
- swapOutBoxes
 - maxsum::util::PostOffice, 62
- UnknownAddressException
 - maxsum::UnknownAddressException, 65
- UnknownVariableException
 - maxsum::UnknownVariableException, 66
- ValIndex
 - maxsum, 19
- ValType
 - maxsum, 19
- validateRange
 - maxsum::DomainIterator, 46
- varBegin
 - maxsum::DiscreteFunction, 40
- varEnd
 - maxsum::DiscreteFunction, 40
- VarID
 - maxsum, 19
- what
 - maxsum::BadDomainException, 30
 - maxsum::DomainConflictException, 41
 - maxsum::EmptyNoticeException, 47
 - maxsum::InconsistentDomainException, 48
 - maxsum::NoSuchElementException, 55
 - maxsum::OutOfRangeException, 56
 - maxsum::UnknownAddressException, 65
 - maxsum::UnknownVariableException, 66